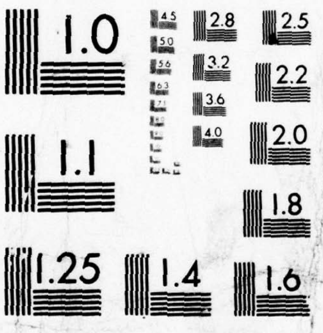
AD-A069 437   NAVAL SURFACE WEAPONS CENTER DAHLGREN LAB VA   F/G 9/2
CONFIGURATION DESCRIPTION LANGUAGE PROCESSOR DESIGN DISCLOSURE.(U)
OCT 78 D J LEMOINE
UNCLASSIFIED   NSWC/DL-TR-3881   NL

1 OF 1
AD
A069437

END
DATE
FILMED
7-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NSWC TR-3881 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* 6 CONFIGURATION DESCRIPTION LANGUAGE PROCESSOR DESIGN DISCLOSURE.		5. TYPE OF REPORT & PERIOD COVERED 9 Final rept.
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Donald J. Lemoine		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Surface Weapons Center ✓ Dahlgren, Virginia 22448		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS N0003079WR92417 9K50TR001
11. CONTROLLING OFFICE NAME AND ADDRESS NSWC (Code K71) Dahlgren, Virginia 22448		12. REPORT DATE 11 October 1978
		13. NUMBER OF PAGES 70
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* 14 NSWC/DL-TR-3881		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT *(of this Report)* 12 70p. Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)* Configuration Description Language, Program Design Language, Prime Structure, Translator, Macro, Stacks, Linked List, Sub-executive		

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The Configuration Description Language System (CDL) was developed to support a complex fire control simulation (TRICS) and the Advanced Weapon System Simulation Program (AWSS). Both of these systems are large simulations which require the facility for modifying the linkage between software components through input directives.

The concepts used in representing program configurations with CDL →(over)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

LB

391 598

79 06 04 099

#20 (Continued)

represent a departure from conventional methods of varying the way in which major program components connect. Program flows are described in a very high level language, through input directives, thus producing configurations that are readable and self-documenting. The system frees the experimenter from having to know the details of the system but at the same time permits the reference and use of actual program components.

The CDL system was developed not only to serve the user but also the programmer. It performs the work normally done by the programmer in interfacing the users' CDL with the actual program.

The CDL system is coded in the SIMSCRIPT II.5 language, version 4.0 and is operational on the CDC 6700 computer system under the SCOPE 3.4 operating system.

FOREWORD

As computer programs grow larger and larger, it becomes increasingly difficult to maintain and use such systems. In the case of simulation models, where it is essential, for experimental purposes, to modify the linkage between software components, it is extremely important to simplify the users' interface with the program system in specifying a program flow (configuration). The Configuration Description Language System (CDL), developed in the Ballistic Sciences Branch of the Computer Programming Division, provides such a tool.

Funding was provided by the Strategic Systems Project Office (SSPO) under project number: N0003079WR92417.

This report was reviewed by Ira V. West, Head of the Ballistic Sciences Branch of the Computer Programming Division.

Released by:

[illegible]

RALPH A. NIEMANN, Head
Strategic Systems Department

Accession For	
NTIS GRA&I	✓
DDC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist.	Avail and/or special
A	

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Section 1. CDL Processor

1.1 Purpose

Existing methods of managing large computational configurations (flows) lack the primary ingredient of flexibility. Generally, computational flow direction is "hard-wired" within the program body, and although the flow may be controlled by the user (via input options, etc.), unused paths for some given configuration must be included in the total program environment, thus increasing core requirements. Furthermore, modifying paths, deleting old ones, and introducing new ones usually require extensive changes.

As an alternative to this "standard operating procedure", the Configuration Description Language (CDL) allows total control of the program flow external to the actual program environment. CDL permits the referencing of program units at the "sub-executive" level (the CDL is the executive) and thus requires that the program be designed such that sub-executives represent the smallest unit of execution flow control. With the exception of data interfaces, sub-executives can be designed without regard to their position in the program flow. With CDL, program flows can be designed easily, and even some basic logic control structures can be incorporated into the configurations.

The Configuration Description Language Processor is a translator which accepts as input, a desired program flow, and generates a top level executive, to be combined with the actual program system. The generated executive is in the same source language as the actual program system.

The CDL Processor is designed to operate either independently, or in conjunction with a general purpose input preprocessor (see reference 1). The combined system could be applied to almost any program system requiring input processing and external control of program flow.

The term "configuration control", used within this report, refers to the ability to control the collection of modules, and the order in which they execute for a given program system. It is not to be confused with the well-known configuration management definition.

In addition to providing unique input capabilities for the user, the CDL Processor provides the programmer with several facilities to aid in the maintenance of the program system. The programmer defines the "sub-executive" environment, thus dictating what sections of the program can and cannot be referenced in the CDL. The management of program data files, library files, etc. is provided through input specifications and BEGIN/REVERT procedures that are automatically generated.
These procedures are unique to the SCOPE 3.4 operating system for the CDC 6000 series.

For large program systems, the processor can optionally generate directives or procedures to cause the program system to be segmented or overlayed; facilities also unique to SCOPE 3.4 (see reference 2).

The CDL Processor is implemented in SIMSCRIPT II.5 version 4.0 and is intended for use on the CDC 6700 system under SCOPE 3.4. A knowledge of CDC nomenclature is assumed.

1.2 Objectives

Knowing that the techniques of the past would be inadequate in providing the configuration control capabilities required by the TRICS fire control program, the problem had to be approached from a different point of view. Therefore, three objectives were defined.

The first objective was to develop a simple, easy to use language for describing program configurations. Other means of describing a program flow or configuration through input were considered such as connectivity relationships; for example, module A connects to module B. However, these were quickly ruled out because they would not provide the full capabilities of configuration control required. Furthermore, since a configuration is a program flow, the most logical way of describing the flow is through a high level language. By adhering to the basic structures of a standard program design language (PDL), the language would be more familiar to a greater number of people.

The second objective was not only to consider the needs of the user but also the programmer by developing procedures whereby maintenance of the program system utilizing CDL would be reduced. This involved elevating many of the activities normally performed internally at the program level, to a high level external environment. In essence, the objective was to eliminate program changes for new or different configurations.

Finally, design the system as table-driven for "off-the-shelf" use. Knowing that the Advanced Weapon System Simulation (AWSS) program would have the same requirements of configuration control as TRICS, the objective was to insure that the system could be easily adapted to AWSS, as well as other programs requiring configuration control.

1.3 Language Specifications

The CDL allows the designer of a configuration to control the sequence in which computational modules are executed. The language provides immediate documentation and encourages a structured approach to designing configurations.

Configuration level modules specified in the CDL map into sub-executives with regard to the internal structure of the program. To add flexibility to the CDL, various types of sub-executives can be defined. The following sub-executive types are supported:

a. "PHASE" sub-executives mark the beginning of a major section in the system

b. "COMPUTES" sub-executives perform computations only

c. "MACRO" sub-executives symbolize some expanded, predefined CDL

d. "DECISION" sub-executives are used for decision making in the CDL

Sub-executive names specified in the CDL can be up to thirty (30) characters in length, the first 10 of which must be unique.

The Configuration Description Language (CDL) is simple, easy to learn, and provides all the capabilities for describing a configuration. Although CDL input is in free format, indentation is encouraged to emphasize structure. The only requirement is that at least 1 blank appear between fields. In the description below, brackets ({}) refer to optional keywords and slash (/) means "or".

The first CDL structure provides a means of defining a major phase of the CDL. It signifies that all modules referenced up to the next phase definition are part of this phase. Its format is:

```
START phase
```

For example:

```
START PATROL
```

Only "PHASE" sub-executives can be referenced with START.

Computational, or macro type sub-executives can be initiated with the RUN command.

```
RUN modules
```

For example:

```
RUN JOE
```

More than one module can be included on a RUN statement, e.g.,

```
RUN JOE TOM
```

When a macro sub-executive is referenced, the CDL Processor will substitute the predefined CDL for this macro and begin processing.

In the following example:

```
START A
   RUN JOE TOM
START B
   RUN FRED JOHN
```

the modules JOE and TOM belong to phase A while FRED and JOHN belong to phase B.

The IF statement is used to determine whether a specified decision module is true or false when invoked, or to test program variables. Control continues with the succeeding statements for the true condition and transfers to the corresponding ELSE statement for the false condition. The ENDIF statement marks the point where the two paths meet.

```
IF condition {{IS} TRUE/FALSE} {THEN}
```

where "condition" is a decision module (sub-executive) or a logical expression; for example,

```
IF TIME < 10.0
```

Either variables global to the system or constants can be tested and up to thirty (30) characters can be used for each expression (no imbedded blanks). This implies that a statement such as

```
IF TIME+3*X < A
```

is legal.

Relational operators available are:

<	less than
>	greater than
>=	greater than or equal to
=	equal to
<=	less than or equal to
∧=	not equal to

"IF" statements can be nested as desired and the "true" condition is assumed as default. They can be of the form "IF THEN" or "IF THEN ELSE". In any case, the IF must be terminated by an ENDIF. For example:

```
IF A IS TRUE THEN
    RUN B
ELSE
    RUN C
ENDIF
RUN D
```

An attempt to reference sub-executive types other than "decision" will result in an error message.

The DO WHILE clause is used to control iterations as long as the specified condition is true (default) or false.

```
DO WHILE condition {{IS}  TRUE/FALSE}
```

where "condition" is the same as with IF statements. An ENDDO designates the end of a segment to be executed repeatedly. For example:

```
DO WHILE Z IS FALSE
    RUN X
    RUN Y
ENDDO
```

The decision module is invoked, or the variables are tested at the beginning of the loop. DO WHILE statements can be nested as desired and IF statements can be included within a DO WHILE segment.

A variation of the DO WHILE statement is the DO FOR statement. It allows the user to specify how many times a segment is to be repeated.

```
DO FOR I = A TO B
```

where I is either a user-defined variable or a predefined program monitored variable. A monitored variable implies that the executive controller will maintain the current value of the variable as the loop progresses, as well as invoke a predefined monitored routine at the beginning of the loop. A and B are either integer constants or integer variables. For example:

```
DO FOR I = 1 TO NMSLS
    RUN A B C
ENDDO
```

For unconditional branching to a specified label, the GO TO statement is provided.

```
GO TO label
```

A statement label identifies a transfer point for a GO TO statement and can appear anywhere in the CDL. A label can be any combination of up to 8 nonblank characters enclosed in apostrophes. For example:

```
'PERFORM'
'10.3'
'ERROR.10'
```

Comments are also allowed in the CDL and are always preceded by two apostrophes (i.e., ''). When these two characters are encountered in the input text, the remainder of the card image is assumed to be comment text.

The TEXT feature of CDL allows the specification of card images to be included in the generated executive. The card images are not modified by the CDL processor and will appear in the same relative position in the executive as the CDL. The TEXT feature is a convenient way of manually inserting code in the executive. The format is:

```
TEXT

     card images

ENDTEXT
```

Individual card images can be specified as text by placing an ampersand (&) in column 1. In this case, the card image is shifted 1 column to the left when included in the generated executive. This is necessary to allow FORTRAN comments to be specified in this manner.

When inserting code with the TEXT feature, the user should consider the impact on the global data environment. Also, caution should be exercised when inserting code that would affect the flow of the executive.

All configurations must be terminated by an "END" statement, e.g.,

```
START A
DO WHILE Z
    RUN X
    RUN Y
ENDDO
END
```

Section 2. CDL Processor Design Concepts

2.1 General Description

The CDL Processor accepts as input, a CDL specifying a particular computational flow. The primary function is to translate the CDL into some desired target source language which is the same as the language used in implementing the program computations. The translated CDL is now the executive (or driver) which can be compiled and linked with the sub-executives referenced in the original CDL (see figure 1). By defining the entire system as a library, only those modules referenced or required by the sub-executives need to be loaded.

Translating the CDL to the target source language is a two-step process. First, the processor builds what is called the FLOW vector from the original CDL source. The FLOW vector contains: a) pointers to data which are essential in the translation; and b) branching information reflecting the flow of the resulting program. Second, the FLOW vector is translated to the target source language. This is a relatively simple process since the FLOW vector contains the information essential in building the logic of the resulting program. The translated source code generally consists of a main program and an executive controller which is called by the main program.

There are three global variables which must be manipulated by the sub-executives. These variables are ERRTYP, ERRNO, and LRESLT and are used in error detection code, and code generated for decision-type sub-executives. ERRTYP denotes whether the detected error is fatal or nonfatal and must be set to one of the following hollerith strings: "FATAL", or "NONFTL". The number of the error detected is stored in ERRNO. Decision-type subexecutives must return a logical value. This is simulated by storing a 0 in LRESLT for false or a 1 in LRESLT for a true result.

2.2 CDL Specification

The CDL processor is capable of operating in conjunction with an input preprocessor (see reference 1). The input preprocessor (INPUTP) is capable of handling a default file and a file of override data. The CDL is always specified on the override data file.

The CDL specification can reference a predefined CDL on a separate file (USE), a CDL from the previous case (PREVIOUS), no CDL (NONE), or a CDL can be created in the override data (CREATE).

Since the CDL processor and input preprocessor are two separate programs, the CDL processor is responsible for separating the CDL specifications and the normal override data into two "created" files. This gives the user the ability to specify all override data on one file in addition to alleviating the input preprocessor of having to sift through CDL specifications.

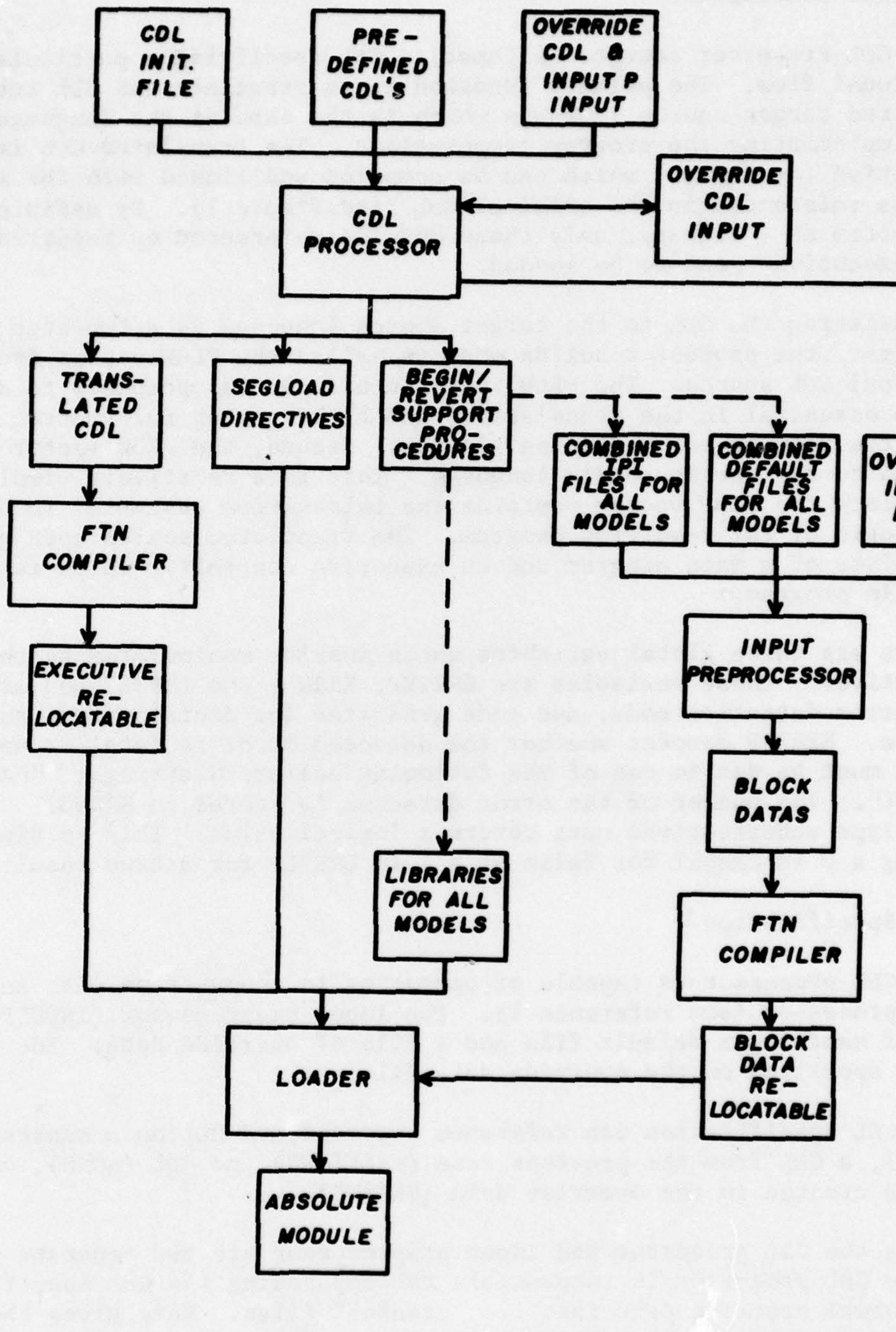


Figure 1. CDL Processor Environment

Hence, on the first case, the CDL processor creates two files, one containing the CDL specifications, the other containing the override data. The CDL processor then reads from the created file. The input preprocessor always reads from the created file, if one was created.

2.3 Initializing the CDL

The CDL processor is essentially table-driven in that all sub-executive names and their types are defined externally on an initialization or "template" file. This file also contains other information required to translate the CDL to some target source language. It is intended to be a programmer-maintained file since it defines the program environment.

Other information on this file includes the name of the resulting main program, the name of the subroutine executive controller, monitored variables and their routines, macro expansions for sub-executives of that type, frontend routines and rearend routines to be called before and after the executive controller, files associated with the models, relational operators used in translating, FORTRAN common blocks (if the target language is FORTRAN) required by the executive controller, and a list of all common blocks required for automatic program segmentation (see reference 2). The relational operators include not only those used in the CDL but also the target language equivalent. For example, "<" would map into ".LT." for FORTRAN.

Keywords are used to designate the varous types of data on this file. Keywords must begin in column 1 while all other data begins beyond column 1 in free format. The layout of this file and an example is given in Appendix A.

Keywords and their associated data can usually be specified in any order, however, there are some restrictions. Data describing the sub-executives to be referenced must be defined before macro expansions, model and file definitions, and frontend and rearend routines. This is necessary since data structures created at the time this keyword is processed are used for processing other keywords. Model and file definitions should be defined before frontend and rearend routines for the same reason. Main program commons, executive controller commons, loader directives, and segment commons must be given in that order and after all other data on the initialization file. This is required since the processor does not store this data but merely copies the card images up to the next keyword when appropriate.

2.4 Basic Data Structures and Table Representation

When processing the CDL, it is necessary to search several different tables to determine what should be stored in the FLOW vector. These tables are either represented by or linked to one common data structure so that the same searching mechanism can be used for table lookup. Since the CDL processor is implemented in SIMSCRIPT II.5, all data structures are defined with SIMSCRIPT syntax.

Basically, table entries are represented by temporary entities filed in a set, where a set exists for each table class. The basic data structures are defined as follows:

```
    EVERY CLASS OWNS AN INPUT.LIST

    EVERY PARAM HAS
        A PNAME,
        A PNAME1,
        A (PIVALUE,PAVALUE),
        A PTYPE,
        A PMDL.REF,
        A PMDL.TYPE,
        A PMDL.ENTITY,
        A POVERLAY,
        AND BELONGS TO AN INPUT.LIST
        AND MAY BELONG TO THE SBX.LIST
```

A CLASS entity exists for each table defined.

The attributes of the PARAM entity are used differently for each table type and are defined for each table type described below.

PARAM entities are also used for storing intermediate data associated with the FLOW vector.

2.4.1 Table Environment

As the CDL Processor scans the given CDL, it must search appropriate tables to determine what actions are to be taken and what data structure pointers are to be stored in the FLOW vector. For example, when scanning a statement such as:

```
    RUN A
```

the processor must search the list of keywords to determine that the "RUN" keyword processor must be invoked. Next, the list of sub-executives must be searched to get the PARAM entity created for the sub-executive A, and then the entity is stored in the FLOW vector.

2.4.1.1 Externally Defined Tables

To process a CDL, it is necessary to know the namesoof all sub-executives to be referenced in the CDL. Sub-executives, their types, routine names, and their associated models are defined on the initialization file.

Every sub-executive is represented by a "PARAM" entity with the attributes defined as follows:

PNAME	=	pointer to the sub-executive name
PNAME1	=	first 10 characters of the name
PAVALUE	=	corresponding routine name (or MACRO entity for PTYPE = MACRO, see below)
PTYPE	=	type of sub-executive: phase, decision, computational, or macro
PMDL.REF	=	pointer to the name of the model with which it is associated
PMDL.TYPE	=	denotes whether this entry is a model or not = blank, not a model = MODEL, entry is a model
PMDL.PTR	=	pointer to the MODEL entity if PMDL.TYPE = MODEL
POVERLAY	=	overlay number for this sub-executive assigned by the CDL Processor

For "macro" type sub-executives, a CDL expansion is implied. The macro expansions are also part of the initialization file and are represented as:

```
EVERY MACRO OWNS A MA.EXPANSION

EVERY CDL.STATEMENT HAS
     A CARD.IMAGE,
     AND BELONGS TO A MA.EXPANSION
```

The MACRO temporary entities are stored in the PARAM temporary entity so that the expansion can be easily retrieved and processed when macro-type sub-executives are referenced. MACRO sub-executives may be specified within a MACRO expansion thus allowing multi-level expansions.

Models and their associated files are specified so that the proper file environment can be defined prior to execution. Associated with each model are libraries, relocatable block data files, local files, data files, input preprocessor initialization (IPI) files, and the default data file. Local files refer to all file names which must be included on the PROGRAM card for CDC FORTRAN EXTENDED programs (includes the LFN for all data files as well).

Since these files are included in a generated BEGIN/REVERT procedure (optional) for executing the model(s), linked lists must be defined for easy retrieval. When models, or sub-executives of a particular model are specified in the CDL, the data structures for the model are saved in another list. When the CDL is processed, files for each model are retrieved for those models in the CDL. These files can be overridden through the the override input. The required data structures are given below.

```
EVERY MODEL HAS
     A MDL.NAME,
     A MNAME1,
     AND OWNS A MDL.LIB.LIST,
                MDL.BLK.LIST,
                MDL.PLO.LIST,
                MDL.EXT.LIST,
                MDL.IPI.LIST,
                MDL.DEF.LIST,
                MDL.FE.LIST,
                MDL.RE.LIST,
     AND BELONGS TO THE MDL.COL.LIST

EVERY LIB.FILE HAS
     A LIB.PFN,
     A LIB.USERID,
     A LIB.LFN,
     A LIB.CYCLE
     AND BELONGS TO A MDL.LIB.LIST

EVERY BLK.FILE HAS
     A BLK.PFN,
     A BLK.USERID,
     A BLK.CYCLE
     AND BELONGS TO A MDL.BLK.LIST

EVERY PLØ.FILE HAS
     A PLØ.LFN,
     A PLØ.BUFSIZE,
     A PLØ.EQ,
     AND BELONGS TO A MDL.PLØ.LIST
     AND BELONGS TO THE PGM.CRD.LIST

EVERY EXT.FILE HAS
     AN EXT.LFN,
     AN EXT.PFN,
     AN EXT.USERID,
     AN EXT.CYCLE
     AND BELONGS TO A MDL.EXT.LIST

EVERY IPI.FILE HAS
     AN IPI.PFN,
     AN IPI.USERID,
     AN IPI.CYCLE
     AND BELONGS TO A MDL.IPI.LIST
```

```
EVERY DEF.FILE HAS
     A DEF.PFN,
     A DEF.USERID,
     A DEF.CYCLE
     AND BELONGS TO A MDL.DEF.LIST

EVERY SPGM HAS
     A SPGM.NAME
     AND MAY BELONG TO
          A MDL.FE.LIST
          A MDL.RE.LIST
          A PGM.LIST
```

Model entities are created for each model specified on the CDL initialization file and are stored in the PMDL.ENTITY attribute of the PARAM entity created for the model.

When the "MODEL" keyword is encountered in the initialization file, it is necessary to search the list of sub-executives to determine if a PARAM entity exists for a model. It is necessary to explicitly define a model as a sub-executive; however the model can be any sub-executive type desired.

Files that are global to the system can be defined by specifying "GLOBAL" as the model name followed by the global files. In this case, PARAM and MODEL entities are created to represent the global model. The PARAM entity is filed in the INPUT.LIST set for sub-executives with the PMDL.ENTITY attribute defined as the MODEL entity.

Monitored variables referenced in the CDL on a "DO FOR" statement are defined on the initialization file. The associated monitored routine name must also be included for each variable. The PARAM entity attributes for monitored variables are defined as:

PNAME	= pointer to the monitored variable name
PAVALUE	= monitored routine name

Frontend or rearend routines can be either global or model-specific and are represented by SPGM entities where SPGM.NAME contains the name of the routine. The SPGM entities are filed in the MDL.FE.LIST set for front-end routines, or the MDL.RE.LIST set for rearend routines referenced by the global MODEL entity or the specific MODEL entity.

Logical operators and their target source language equivalent are also part of the initialization file and are represented by PARAM entities:

PNAME	= pointer to the logical operator symbol(s)
PAVALUE	= target source language equivalent

2.4.1.2 Internally Defined Tables

Keywords for the CDL Initialization file are defined in a table internally. The attributes of the PARAM entity are defined as follows:

PNAME	= pointer to the keyword name
PAVALUE	= keyword index number

The keywords currently available are:

1. SUB-EXECS
2. MACROS
3. MONITORED
4. MODELFILES
5. FRONTEND
6. REAREND
7. OPERATORS
8. LANGUAGE
9. PROGRAM
10. CONTROLLER
11. COMMONS
12. SEGMENT
13. EXEC
14. LOADER
15. RECOVERY

CDL keywords are defined internally, again by PARAM entities. Each keyword has an associated keyword routine. Attributes of the PARAM entity are defined as follows:

PNAME	= pointer to the keyword name
PIVALUE	= subprogram variable for the keyword routine

The following is a list of CDL keywords currently supported:

Keyword	Associated Subprogram
RUN	RUN.KEYWORD
IF	IF.KEYWORD
ELSE	ELSE.KEYWORD
ENDIF	ENDIF.KEYWORD
DO	DO.KEYWORD
ENDDO	ENDDO.KEYWORD
START	START.KEYWORD
&COMMENT&	COMMENT.KEYWORD
&LABEL&	LABEL.KEYWORD
TEXT	TEXT.KEYWORD
GO	GO.TO.KEYWORD
END	END.KEYWORD
ENDTEXT	TEXT.KEYWORD

The keywords COMMENT and LABEL are preceded by the symbol "&" to denote that they are detected by special symbols in the CDL text rather than the names given.

FILE keywords represent those keywords available for file specifications on the initialization file, or for overriding files given on the initialization file (see section 2.6.2). A PARAM entity is created for each keyword and files in the INPUT.LIST set for file keywords. The file keywords currently available are:

1. LIBRARY
2. BLOCK.DATA
3. LOCAL
4. DATA
5. IPI
6. DEF

2.5 CDL Runtime Options

There are four runtime options for the CDL Processor which are specified through the PARM feature of SIMSCRIPT. The options are included on the "execute" card primarily for convenience. The four options are:

LOAD	= Option for loading "ØVL" perform overlay loading "SEG" perform SEGLOAD, i.e., generate segment loader directives "NORM" perform normal loading (default)
MACRO	= Macro expansion printout option "YES" for macro expansion printout, "NO" for no macro expansion printout (default)
JCL	= Generate BEGIN/REVERT procedure option "YES" to generate procedures "NO" to suppress the generation (default)
LC	= last column for all input to the CDL Processor (default is 72)

2.6 Override Input

2.6.1 CDL Specification

CDL specifications can only appear in the override input, and can be given in any order. There are four CDL specification options available:

1. USE
2. CREATE
3. PREVIOUS
4. NONE

USE implies that a CDL from a file containing a catalog of CDL's will be used. This file must have CDL's defined as follows:

```
CDL name 1
    (CDL specifications)
    .
    .
    .
    END
CDL name 2
    (CDL specifications)
    .
    .
    .
```

where the CDL keyword begins in column 1. To retrieve a CDL from the file, one would include in the override input:

```
CDL
    USE name
```

and the CDL keyword must begin in column 1.

CREATE means that CDL specifications will be included in the override input, for example,

```
CDL
    CREATE
    RUN A
    RUN B
    END
```

The keyword PREVIOUS following CDL in the override input means use the CDL from the previous case. This condition is default on the second and succeeding cases and it cannot be specified on the first case.

If no CDL is required, then the keyword NONE <u>must</u> be specified.

2.6.2 File Override Data

Although the file environment for each model is defined on the CDL initialization file, it is sometimes desirable to change some of these files. Libraries, relocatable block data routine files, data files, IPI, local, and default files for any model can be overridden through specifications on the override input file. To do this, the FILES keyword is required (beginning in column 1 of course) followed by the model name, and the files to add or delete. The general form is as follows:

```
FILES
     model name
      action type PFN  LFN   user identification   cycle number
     END
      .
      .
      .
```

where "action" is ADD or DELETE, "type" is LIBRARY, BLOCK.DATA, DATA, IPI, LOCAL, or DEF, "PFN" is the permanent file name, "LFN" is the local file name (specified for DATA, LIBRARY, and LOCAL files only). PFN and user identification are not specified for LOCAL type files.

To change this environment internally, the INPUT.LIST set for models is searched for the given model. When found, files are either added by creating a file entity (LIB.FILE, BLK.FILE, PLO.FILE, EXT.FILE, IPI.FILE, or DEF.FILE) and filing in the appropriate set, or removing entities when files are deleted.

2.7 Summary of Files Associated With the CDL Processor

The following table describes the input/output files associated with the CDL processor.

Unit Number	Type	Description
SIMU2	OUTPUT	Dummy output file
SIMU3	OUTPUT	Created INPUTP override input file
SIMU5	INPUT	Original override input file
SIMU6	OUTPUT	Standard output file
SIMU31	INPUT	CDL initialization file
SIMU33	INPUT	File of predefined CDL's
SIMU35	INPUT/OUTPUT	Created CDL override input file
SIMU37	OUTPUT	BEGIN/REVERT procedure file
SIMU39	OUTPUT	Generated main program and executive controller (in the target source language)
SIMU41	OUTPUT	Segment directives file

Section 3. Translating the CDL to High Level Source Code

3.1 FLOW Vector Generation

The phase of the CDL processor that translates the CDL to the FLOW vector works using the stack principle. Control structures such as IF and DO are placed on the stack to allow nested structures and also for storing information about the structure which is eventually transferred to the FLOW vector. Also, the stack principle simplifies the verification of the structure syntax.

The data structure used to represent the stack and its entries is as follows:

```
EVERY STRUCTURE HAS
      A STR.NAME,
      A STR.ENTITY,
      AND BELONGS TO THE STR.STACK

DEFINE STR.STACK AS A LIFO SET
```

STR.NAME is the name of the structure (IF or DO), STR.ENTITY points to a temporary entity containing information about the structure and its relation to the FLOW vector, and STR.STACK is a LIFO (Last In, First Out) set representing the stack.

The translation phase involves reading tokens from the file containing the CDL, classifying the tokens, and invoking the appropriate keyword processing routine. Some keyword routines read other tokens required for the particular structure. For example, the RUN keyword requires that a token representing a sub-executive be read.

When scanning and processing the CDL, it is necessary to save models and sub-executives referenced. This is required when it is necessary to generate BEGIN/REVERT procedures to attach the files for each model, or when it is required to generate segment loader directives for the sub-executives.

As sub-executives are encountered in the CDL on RUN, START, IF, etc. statements, the PMDL.REF attribute of the PARAM entity for the sub-executive contains the associated model. If the model already exists in the list (MDL.CDL.LIST), then no action is required. However, if it does not exist, the sub-executive list is searched for the associated model. When found, the MODEL entity (stored in PMDL.PTR) can be filed in the MDL.CDL.LIST set.

A similar procedure is followed when collecting sub-executives for the segment loader directives. If the sub-executive already exists in the list (SBX.LIST), no action is required. Otherwise, it is filed in the SBX.LIST set.

When macros are encountered in the CDL text, it is necessary to save the current card image being processed and the last column read. This is done so that processing of this card image can resume when the processing of the macro is complete. Card images for the macro are represented by CDL.STATEMENT entities in the MA.EXPANSION set for the macro. Processing for the macro terminates when the last entity in the MA.EXPANSION set is processed. When the processor encounters a macro while processing a macro, the current macro is preempted and its processing state (card image and last column read) is preserved on a stack. The data structure for this stack is as follows:

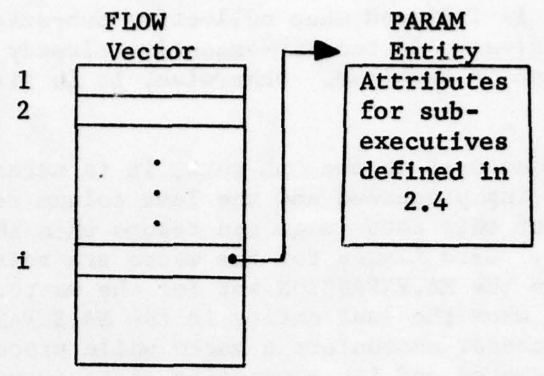
```
EVERY STACKED.MACRO HAS
     A SM.CDL.STATE,
     A SM.MACRO,
     A SM.COLUMN,
     A SM.FIRST.CARD,
     AND BELONGS TO THE STACK.OF.MACROS
```

where SM.MACRO is the MACRO entity, SM.ENTITY is a pointer to the current CDL.STATEMENT entity being processed, SM.COLUMN is the last column read, SM.FIRST.STATE denotes whether this is the first card image of the macro or not (YES or NO), and STACK.OF.MACROS is the stack. When processing of the new macro completes, the macro previously preempted is removed from the stack, and its processing resumes.

Each structure of the CDL defined in section 1.2 requires that certain information be stored in the FLOW vector. This information, the required data structures, and stack operations for each keyword are described below.

3.1.1 RUN Keyword

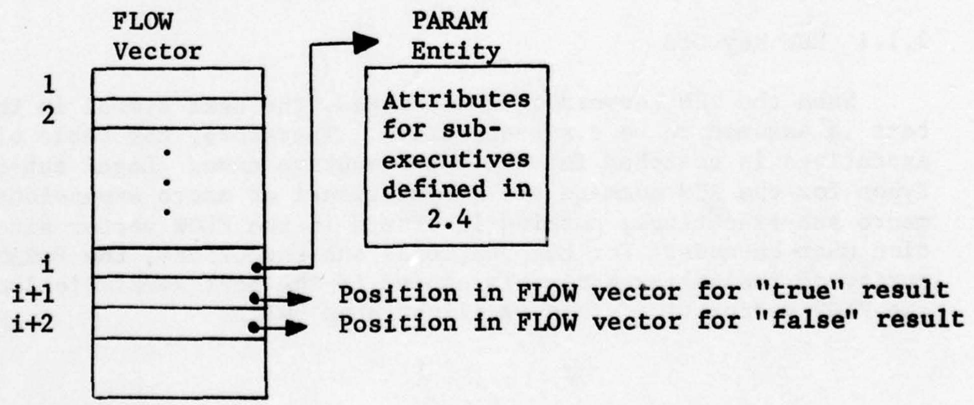
When the RUN keyword is encountered, the next symbol in the input text is assumed to be a sub-executive. Therefore, the table of sub-executives is searched for the sub-executive name. Legal sub-executive types for the RUN command are computational or macro expansions. For macro sub-executives, nothing is stored in the FLOW vector since a substitution must be made. For computational sub-executives, the PARAM temporary entity of the sub-executive is stored in the next available location in the FLOW vector (i,. This is illustrated below.



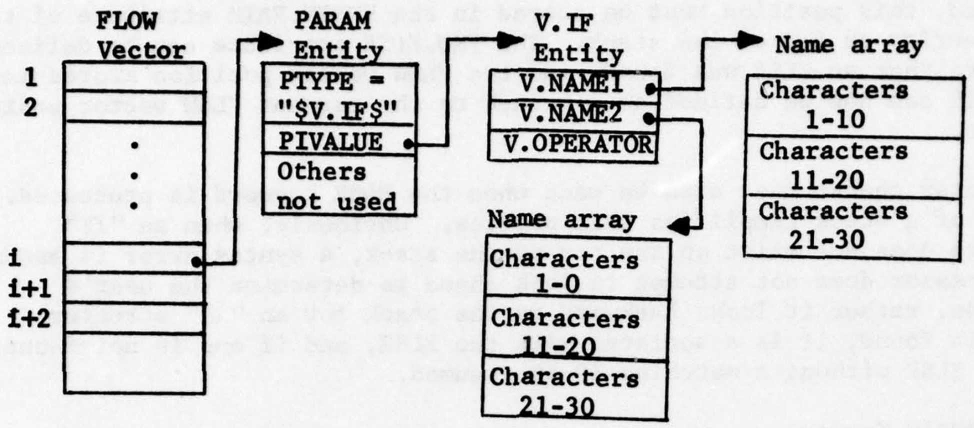
3.1.2 IF Keyword

Since IF can be used with both decision modules and program variables, the token following the IF determines what will be stored in the FLOW vector. In either case, three positions in the FLOW vector are defined: the pointer to the appropriate entity, the position (negated) in the flow vector to branch to if the condition is true, and the position (negated) in the FLOW vector to branch to if the condition is false. It is not possible for the processor to determine one of these positions until the ELSE for the IF has been processed. If the "true" condition is assumed as default (i.e., the clause "IS TRUE" is specified, or omitted), then the "true" position is merely the current counter plus 2, likewise, if the "false" condition is specified, the "false" position is the same. The remaining position is defined when the "ELSE" is encountered.

For sub-executives, the following illustrates what is stored in the FLOW vector.


FLOW
Vector
1
2
i
i+1
i+2
PARAM
Entity
Attributes for sub-executives defined in 2.4
Position in FLOW vector for "true" result
Position in FLOW vector for "false" result

When a variable is specified, the following illustrates the linked structures required.



The V.IF entity is created for storage of information about the variables specified in the IF. It is defined as follows:

```
EVERY V.IF HAS
      A V.NAME1,
      A V.NAME2,
      A V.OPERATOR
```

A new data structure must be created and placed on the structure stack so that the correct paths can be defined when the "ELSE" and "ENDIF" are encountered. This data structure is:

```
EVERY CDL.IF HAS
      AN ELSE.PATH,
      AN ENDIF.PATH,
      AN IF.LINE.NO,
      A FND.ELSE
```

Where ELSE.PATH is the position in the FLOW vector containing the "false" path. ENDIF.PATH will contain the position in the FLOW vector following the positions up to the "ELSE" where a branch to the ENDIF must be defined when ENDIF is encountered. IF.LINE.NO is merely the CDL line number and FND.ELSE is used to denote when an ELSE was encountered for error detection of multiple "ELSE" keywords for the same IF.

3.1.3 ELSE Keyword

The current position in the FLOW vector must be defined as a branch to the ENDIF, but since this won't be available until the ENDIF is encountered, this position must be stored in the ENDIF.PATH attribute of the CDL.IF entity on top of the stack. The FND.ELSE attribute can be defined to denote that an ELSE was found, and the FLOW vector position stored in ELSE.PATH can now be defined as a branch to the current FLOW vector position plus one.

Syntax checks must also be made when the ELSE keyword is processed. The use of a stack simplifies this process. Obviously, when an "IF" structure does not exist on the top of the stack, a syntax error is assumed. The processor does not attempt to look ahead to determine the user's intention, rather it looks backward on the stack for an "IF" structure. If one is found, it is associated with the ELSE, and if one is not found, then an ELSE without a matching IF is assumed.

3.1.4 ENDIF Keyword

No additional positions in the FLOW vector are required for this keyword. The FLOW vector position defined in ENDIF.PATH on the stack can now be defined as a branch to the current FLOW vector position. This completes the processing for the "IF" structure, the stack is popped and all data structures destroyed.

The following example illustrates what is stored in the FLOW vector for an "IF THEN ELSE" structure.

```
IF A IS TRUE
    RUN B
ELSE
    RUN C
ENDIF
```

The following FLOW vector results:

```
FLOW(1) = PARAM entity of sub-executive A
FLOW(2) = -4 (true path negated)
FLOW(3) = -6 (false path negated)
FLOW(4) = PARAM entity of sub-executive B
FLOW(5) = -7 (branch to ENDIF)
FLOW(6) = PARAM entity of sub-executive C
```

An examination of the stack is done to determine if a syntax error has occurred. If the top of the stack is not an "IF", then the stack is scanned until an "IF" is found or the stack is exhausted. The existence of an "IF" requires its removal from the stack. When no "IF" is found, the ENDIF is assumed to be extraneous.

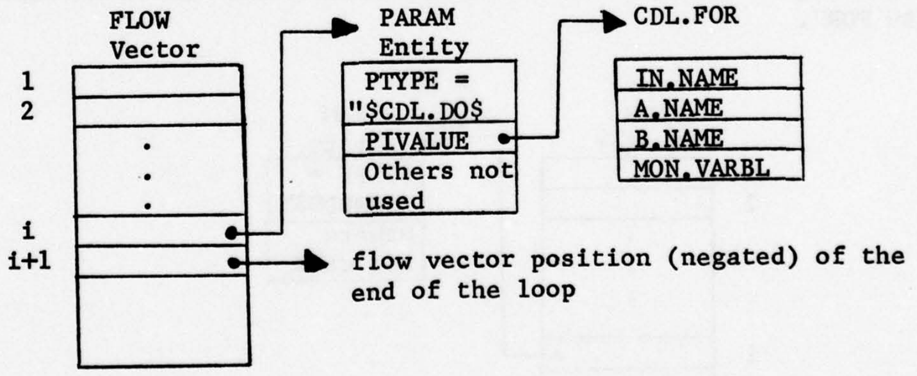
3.1.5 DO FOR Keyword

This keyword also requires a linked list to be set up in the FLOW vector to properly store the data needed. Two positions in the FLOW vector are defined. The first position is set to a PARAM entity defining the structure type and a pointer to another entity holding information about the "DO" structure. This entity is defined as:

```
EVERY CDL.FOR HAS
    AN IN.NAME,
    AN A.NAME,
    A B.NAME,
    A MON.VARBL
```

Where IN.NAME is the name of the loop index, A.NAME is the initial value of the loop and B.NAME is the last value of the loop. MON.VARBL is the name of the monitored routine if one exists.

The second position in the FLOW vector is the FLOW vector position to branch to when the loop terminates. The following illustrates what is stored in the FLOW vector.



A new type data structure must be created and placed on the structure stack so that the correct path can be defined when the "ENDDO" is encountered. This data structure is:

```
EVERY CDL.DO HAS
    A DO.LABEL,
    AN ENDDO.PATH,
    A DO.LINE.NO,
    A DO.TYPE
```

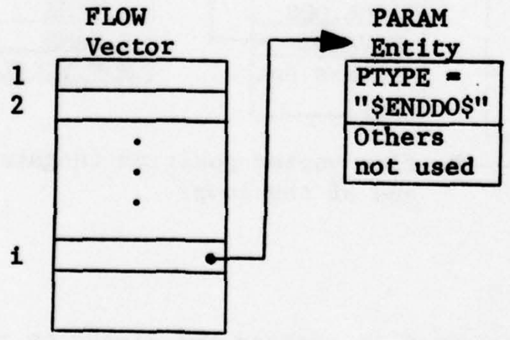
Where DO.LABEL contains the FLOW vector position of the "DO" origin, ENDDO.PATH contains the position in the FLOW vector which will be defined as a branch to the end of the DO, DO.LINE.NO is the CDL line number, DO.TYPE is the type of the DO, "FOR" or "WHILE".

3.1.6 DO WHILE Keyword

This keyword is treated the same way as an "IF" keyword since what follows "DO WHILE" is identical. The only difference is in the stack processing where a CDL.DO entity is stored in STR.ENTITY rather than a CDL.IF entity. The three positions in the FLOW vector are defined as in an "IF" with the exception of the "false" path position defined as a branch to the end of the "DO".

3.1.7 ENDDO Keyword

This keyword defines the next available position in the FLOW vector to a PARAM entity with the PTYPE attribute set to "$ENDDO$" for the "DO FOR" structure. For ENDDO's associated with "DO WHILE" structures, the FLOW vector position contains a branch to the beginning of the DO. This is necessary since some high level languages do not have a true "DO WHILE" feature and the generated code for the ENDDO is generally a branch to the beginning of the DO. The following illustrates the PARAM entity for the "DO FOR".



No other information needs to be stored with the structure. Even if a label is required to terminate the loop (as in FORTRAN), the FLOW vector index, i, would be used.

The FLOW vector position defined in the ENDDO.PATH attribute of the CDL.DO entity stored on the top of the stack can now be defined as a branch to the current FLOW vector position (the end of the DO). For the "DO FOR" keyword, it is the second word reserved when the "DO" keyword was encountered. For the "DO WHILE" keyword, the false path position is defined as a branch to the FLOW vector position of the end of the DO plus 1 (so that a branch to the outside range of the DO is effected).

The following example shows the FLOW vector positions defined with the "DO FOR" keyword.

```
DO FOR I = 1 TO 10
     RUN A
     RUN B
ENDDO
RUN C
```

This would produce the following FLOW vector.

```
FLOW(1) = PARAM entity holding CDL.FOR entity
FLOW(2) = -5
FLOW(3) = PARAM entity for sub-executive A
FLOW(4) = PARAM entity for sub-executive B
FLOW(5) = PARAM entity for the ENDDO
FLOW(6) = PARAM entity for sub-executive C
```

The following example shows the FLOW vector positions defined with the "DO WHILE" keyword.

```
DO WHILE Z IS TRUE
     RUN A
     RUN B
ENDDO
RUN C
```

The following FLOW vector would result.

```
FLOW(1) = PARAM entity for sub-executive Z
FLOW(2) = -4 (true path)
FLOW(3) = -7 (false path)
FLOW(4) = PARAM entity for sub-executive A
FLOW(5) = PARAM entity for sub-exeuctive B
FLOW(6) = -1 (branch to DO origin)
FLOW(7) = PARAM entity for sub-executive C
```

When the "IS FALSE" clause is used, the values of the true and false paths are switched.

The required stack operation for the ENDDO is to remove the first structure from the stack (pop the stack). If the first structure is not a DO, a syntax error results, and an attempt is made to locate a DO on the stack. If one is found, the DO is removed from the stack. If a DO is not found, the ENDDO is assumed to be extraneous.

3.1.8 START Keyword

This keyword is essentially the same as the RUN keyword with the exception that only phase type sub-executives are allowed. The PARAM entity, when retrieved from the table of sub-executives, is stored in the next available position in the FLOW vector.

3.1.9 GO TO Keyword

Additional data structures are required for labels so that all label references can be satisfied when the entire CDL is processed. These data structures are:

```
EVERY CDL.LABEL HAS
     A NAME.OF.LABEL,
     AN ORIGIN,
     AND OWNS A REF.LIST,
     AND BELONGS TO A SET.OF.LABELS

EVERY REFERENCE HAS
     A REF.INDEX,
     AND BELONGS TO A REF.LIST
```

The current position in the FLOW vector is defined as a branch to the origin of the label. This may not be possible if the origin is not known. If the label does not exist in the SET.OF.LABELS, then a CDL.LABEL entity is created and filed in the SET.OF.LABELS set. If the origin of the label is unknown, then a REFERENCE entity is created and filed in the REF.LIST set with REF.INDEX set to the current FLOW vector position. If the origin is known, then the FLOW vector can be defined as a branch to the origin of the label.

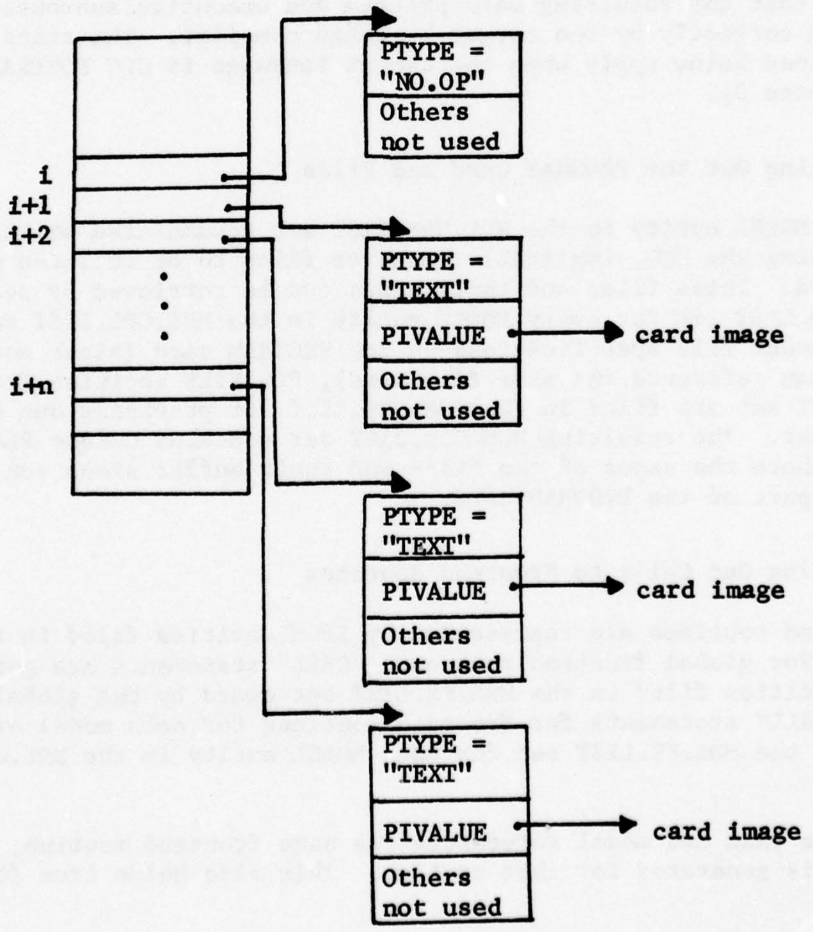
3.1.10 LABEL Keyword

When labels are encountered in the input text, a CDL.LABEL is created if one does not exist. If one does exist, then it is retrieved from the SET.OF.LABELS set. If the ORIGIN is non-zero, then this is a multiplydefined label, producing a syntax error. If the ORIGIN is zero, then the ORIGIN is defined as the current FLOW vector position, and all REFERENCE entities in the label's REF.LIST set can be satisfied.

When the CDL processing is completed, the ORIGIN attribute indicates whether a label is undefined or not.

3.1.11 TEXT Keyword

The TEXT keyword implies that card images up to the ENDTEXT keyword are to be transferred, without modification, to the generated executive; or, in the case where ampersand appears in column 1, individual card images are treated as text. A PARAM entity representing "no operation" is created and stored in the FLOW vector first, followed by a PARAM entity for each card image in the text, with the PIVALUE attribute defined as the pointer to the card image. The "no operation" entities are required during the translation phase to insure that no extraneous code is inserted in the text card images. The contents of the FLOW vector are illustrated below. (Note that, for FORTRAN, text code must begin in column 7 or beyond and not exceed 72.)



3.1.12 END Keyword

A zero is stored in the next available location in the FLOW vector when this keyword is encountered.

3.2 Translating the FLOW Vector

Translating the FLOW vector to the target language is a relatively simple process since there are only two data types stored in the FLOW vector: PARAM entity pointers, and negative entities. PARAM entities can be one of seven different types.

a. Computational sub-executives (PTYPE = "COMPUTES" or "PHASE")
b. Decision sub-executives (PTYPE = "DECISION")
c. Variable IF (PTYPE = "$V.IF$")
d. DO FOR (PTYPE = "$CDL.DO$")
e. ENDDO (PTYPE = "$ENDDO$")
f. TEXT (PTYPTE = "TEXT")
g. No operation (PTYPE = "NO.OP")

In addition to translating the FLOW vector, other information must be written so that the resulting main program and executive subroutine will be compiled correctly by the target language compiler. The translation steps outlined below apply when the target language is CDC FORTRAN Extended (see Reference 3).

3.2.1 Writing Out the PROGRAM Card and Files

Every MODEL entity in the MDL.CDL.LIST set accumulated while scanning and processing the CDL, implicitly requires files to be included on the PROGRAM card. These files and their sizes can be retrieved by searching the MDL.PLO.LIST set for every MODEL entity in the MDL.CDL.LIST set. To avoid redundant file specifications on the PROGRAM card (since more than one model can reference the same file name), PLO.FILE entities from the MDL.PLO.LIST set are filed in the PGM.CRD.LIST set providing one does not already exist. The resulting PGM.CRD.LIST set contains unique PLO.FILE entities, where the names of the files and their buffer sizes can be written as part of the PROGRAM card.

3.2.2 Writing Out Calls to Frontend Routines

Frontend routines are represented by SPGM entities filed in the MDL.FE.-LIST set. For global frontend routines, "CALL" statements are generated for SPGM entities filed in the MDL.FE.LIST set owned by the global MODEL entity. "CALL" statements for frontend routines for each model are generated by scanning the MDL.FE.LIST set for each MODEL entity in the MDL.CDL.LIST set.

If more than one model references the same frontend routine, then only one "CALL" is generated for that routine. This also holds true for rearend routines.

3.2.3 Writing Out the Call to the Executive

This is a simple procedure of just writing out a "CALL" to the executive controller name defined on the CDL initialization file.

3.2.4 Writing Out Calls to Rearend Routines

Rearend routines are represented by SPGM entities filed in the MDL.RE. LIST set. For global rearend routines, "CALL" statements are generated for SPGM entities filed in the MDL.RE.LIST set owned by the global MODEL entity for rearend routines. "CALL" statements for rearend routines for each model are generated by scanning the MDL.RE.LIST set for each MODEL entity in the MDL.CDL.LIST set.

3.2.5 Writing Out the Main Program End

This involves writing out STOP and END to the output file.

3.2.6 Writing Out the Executive Subroutine Header

Given the executive controller name defined on the CDL initialization file, a SUBROUTINE card is generated.

3.2.7 Writing Out FORTRAN Common Blocks

The FORTRAN common blocks defined on the CDL initialization file are not stored internally, they are merely copied from the initialization file to the output file.

3.2.8 Writing Out the Executive Controller from the FLOW Vector

The FLOW vector contains pseudo GO TO's in the form of negative entries referencing FLOW vector array entries. In translating the FLOW vector, these negative entries cause the generation of GO TO statements. Therefore, it is necessary to perform a pre-scan of the FLOW vector and store the absolute value of all negative entries in the LIST.OF.LABELS set. As FORTRAN statements are constructed from each element of the FLOW vector, statement labels are assigned accordingly whenever the top entry in the LIST.OF.LABELS set matches the current element in the FLOW vector.

When scanning the FLOW vector and constructing code, it is first necessary to classify the FLOW vector entry. When the FLOW vector entry is negative, it is a simple matter of constructing a GO TO with the label being the absolute value of the entry. All other entries in the FLOW vector are assumed to be PARAM entities.

If the overlay load option has been selected, a "CALL" to any sub-executive is replaced by a "CALL" to this sub-executive's overlay. A "MAIN" program is generated for each overlay (corresponding to a sub-executive) containing a "CALL" to the sub-executive.

If the PTYPE attribute of the PARAM entity is "$V.IF$", then the PARAM entity forms the linked list described in section 3.1.2. The PIVALUE contains the V.IF entity which holds essential information for translation. The next two positions in the FLOW vector contain the true and false paths respectively. A description of the actual translation is best illustrated by an example.

Assume the following CDL:

```
IF X < Y
     RUN A
ELSE
     RUN B
ENDIF
END
```

The FLOW vector is as follows:

```
FLOW(1) = PARAM entity pointing to the V.IF entity
FLOW(2) = -4
FLOW(3) = -6
FLOW(4) = PARAM entity of sub-executive A
FLOW(5) = -7
FLOW(6) = PARAM entity of sub-executive B
FLOW(7) = 0
```

The following code would be produced (LOAD = NØRM):

```
      IF (.NOT.(X.LT.Y)) GO TO 6
4     CALL A
      IF (ERRNO.NE.O.AND.ERRTYP.EQ.FATAL) RETURN
      GO TO 7
6     CALL B
      IF (ERRNO.NE.O.AND.ERRTYP.EQ.FATAL) RETURN
7     RETURN
      END
```

Notice that error detection code is generated after each call to a sub-executive assuming that the number of the error is stored in ERRNO, and the error type (fatal or non-fatal) is stored in ERRTYP.

For PTYPE equal to "$CDL.DO$", the linked list described in section 3.1.5 is assumed. The CDL.FOR entity, containing the essential information for constituting the loop, is stored in the PIVALUE attribute.

An example of using the DO FOR is as follows:

```
DO FOR I = 1 to 10
     RUN A
     RUN B
ENDDO
END
```

The FLOW vector is as follows:

```
FLOW(1) = PARAM entity pointing to the CDL.FOR entity
FLOW(2) = -5
FLOW(3) = PARAM entity of sub-executive A
FLOW(4) = PARAM entity of sub-executive B
FLOW(5) = PARAM entity for the ENDDO
FLOW(6) = 0
```

Attributes of the CDL.FOR entity would be defined as follows:

```
IN.NAME    = "I"
A.NAME     = "1"
B.NAME     = "10"
MON.VARBL = blank
```

The following code would be generated (LOAD = NØRM):

```
      DO 5 I = 1,10
      CALL A
      IF (ERRNO.NE.O.AND.ERRTYP.EQ.FATAL) RETURN
      CALL B
      IF (ERRNO.NE.O.AND.ERRTYP.EQ.FATAL) RETURN
  5   CONTINUE
      RETURN
      END
```

If a monitored routine were associated with the loop index, then a call to the routine would be generated after the DO statement and error detection code would follow.

When the PTYPE attribute of the PARAM entity obtained from the FLOW vector is equal to "$ENDDO$", the generated code depends upon the type of DO. In the above example, the generated code is merely a "CONTINUE" with the "DO" label to mark the end of the DO. For "WHILE" type loops, a branch to the beginning of the loop is generated.

If the PTYPE attribute is equal to "DECISION", then the PARAM entity represents a decision type sub-executive. It is assumed that decision sub-executives define a global variable (LRESLT) denoting whether it is true (LRESLT = 1) or false (LRESLT = 0). The generated code is essentially a CALL to the sub-executive, error code, and a branch to the true or false path (determined from the two FLOW vector positions following the PARAM entity).

For example:

```
          IF A IS TRUE
               RUN B
          ELSE
               RUN C
          ENDIF
          END
```

would translate to (LOAD = OVL):

```
          CALL OVERLAY(6HA   , 1B, 0, 6HRECALL)
          IF (ERRNO.NE.0.AND.ERRTYP.EQ.FATAL) RETURN
          L = LRESLT + 1
          LRESLT = 0
          GO TO (6,4),L
     4    CALL OVERLAY(6HB   , 2B, 0, 6HRECALL)
          IF (ERRNO.NE.0.AND.ERRTYP.EQ.FATAL) RETURN
          GO TO 7
     6    CALL OVERLAY(6HC   , 3B, 0, 6HRECALL)
          IF (ERRNO.NE.0.AND.ERRTYP.EQ.FATAL) RETURN
     7    RETURN
          END
```

When a FLOW vector entry value of 0 is encountered the "RETURN" and "END" statements are generated and the translation process is complete.

Card images specified in the CDL as "text" are represented by PARAM entities with the PTYPE attribute set to "TEXT". The card image, the pointer of which is stored in PIVALUE, is written verbatim to the translated output file.

Text card images are always preceded by a "no operation" PARAM entity in the FLOW vector. If this entity did not exist, it is possible for a label to be assigned to a card image in the text, thus violating the definition of text cards. "No operation" PARAM entities in the FLOW vector are represented by PTYPE set to "NO.OP". They are translated to a "CONTINUE" statement with a label equal to the current position in the FLOW vector. For example

```
          TEXT
               CALL A
               CALL B
               X = X + 1
          ENDTEXT
          END
```

would translate to:

```
     1    CONTINUE
          CALL A
          CALL B
          X = X + 1
```

Section 4. Overlay Load Generation

The most common way of executing a large program system on the CDC 6700 is by using the overlay load facilities. The program system required by the CDL Processor is composed of a main program, an executive controller, and the sub-executives, such that an overlay structure is naturally formed. The CDL Processor automatically performs all the steps necessary in causing an overlay load.

Currently, the overlay load option is available only for FORTRAN EXTENDED programs. Basically, the main program and executive controller are part of the main overlay and each sub-executive is a primary overlay (see reference 2). There is a slight variation in the code generated for the executive controller causing the overlay to be referenced rather than the sub-executive directly. There are also some slight changes to the generated BEGIN/REVERT Procedures (see section 6).

Section 5. Generation of Segment Directives

CDC segment directives (see reference 2) can be requested when the core requirements of the configuration to be executed become very large. The segment directives generated are the minimum required for the segment loader (for the CDC 6700 SCOPE 3.4 system).

The CDL processor views the entire system as two levels: the root segment (Level 0), and sub-executives (Level 1). The root segment would contain the main program, and executive controller, and the next level (1) would contain the sub-executives. All other modules are considered movable, i.e., the loader assigns each module to a segment that can be accessed by all the segments referencing the module.

This option is specified via the "PARM" parameter when executing the CDL processor.

When the CDL has been processed, the SBX.LIST set contains PARAM entities representing the sub-executives specified in the CDL. Segment directives are generated from this information.

As an example, assume that the following CDL is specified:

```
START A
     RUN B
     RUN C
     RUN D
     RUN E
END
```

and assume that the name of the main program given in the CDL initialization file is MAIN. The following segment loader directives are generated:

```
ROOT    TREE    MAIN
        LEVEL
        TREE    A
        TREE    B
        TREE    C
        TREE    D
        TREE    E
MAIN    GLOBAL    common list
        END
```

The GLOBAL common list must be predefined on the CDL initialization file with all common blocks from all models.

Section 6. BEGIN/REVERT Procedures

An optional feature of the CDL processor is to generate CDC-compatible BEGIN/REVERT procedures that execute the input preprocessor, attach model related files, and generate the load sequence to execute the model(s) specified in the CDL. The generated procedures operate within fixed BEGIN/REVERT procedures that perform functions which are invariant to the models requested.

The first fixed BEGIN/REVERT procedure (EXECUTE) attaches the CDL processor, the master old program library and performs an UPDATE to retrieve the initialization file for the CDL processor. A recursive procedure is invoked (CDLPREPARE) that performs functions required on a case by case basis. This procedure executes the CDL processor and then invokes 3 procedures generated by the CDL processor.

The generated procedure that is invoked first (IPIPREPARE) retrieves the input preprocessor initialization data (IPI) for each model specified in the CDL. The CDL processor assumes that each model has an IPI deck. The generated JCL consists of an attach for each models' IPI file and then a sequence to combine all IPI decks onto one file. The MDL.CDL.LIST set contains MODEL entities of those models referenced in the CDL.

The second generated procedure (FILES) attaches all data files, libraries, and default block data files required for loading and execution. The names of data files to attach are obtained from the EXT.FILE entities which are filed in the MDL.EXT.LIST set for each model. If the same local file name is specified for more than one model specified in the CDL, then the first models' file is used. Libraries are obtained from the LIB.FILE entities filed in the MDL.LIB.LIST set for each model. Default block data routines are obtained from the BLK.FILE entities filed in the MDL.BLK.LIST set for each model.

The third procedure (LDEXEC) executes the input preprocessor, compiles the main program, executive controller, and override block data routines, and performs the load sequence required for execution. The load sequence includes a SEGLOAD if segment loader directives were generated, a load of the main program, executive controller, default block data routines, and override block data routines. When an overlay load is requested, the main program and executive controller are part of the main overlay while each sub-executive is a primary overlay.

An outline of typical BEGIN/REVERT procedures is given in Appendix B.

REFERENCES

1. Peter J. Goyette, *User Guide for INPUTP, General Purpose Input Processor*, NSWC Technical Report 3880, Naval Surface Weapons Center, Dahlgren, Virginia.

2. Control Data Corporation, *Loader Version 1 Reference Manual*, CDC Publication #60344200.

3. Control Data Corporation, *FORTRAN EXTENDED Reference Manual*, CDC Publication #60305601.

APPENDIX A

CDL Initialization File Layout

APPENDIX A

The information on the initialization file is needed to initialize the CDL processor.

The structure of this file is as follows:

```
SUB-EXECS
    GENERIC NAME 1         ROUTINE NAME 1       TYPE       MODEL
    GENERIC NAME 2         ROUTINE NAME 2       TYPE       MODEL
    .                            .                .          .
    .                            .                .          .
    .                            .                .          .
MACROS
    SUB-EXECUTIVE MACRO NAME 1
         (CDL FOR THIS MACRO)
    END
    SUB-EXECUTIVE MACRO NAME 2
         (CDL FOR THIS MACRO)
    END
    .
    .
    .
MONITORED VARIABLES
    NAME 1                 MONITORED ROUTINE 1
    NAME 2                 MONITORED ROUTINE 2
    .                            .
    .                            .
    .                            .
MODELFILES
    MODEL NAME 1 OPTIONAL DESCRIPTION
      LIBRARIES     PFN 1     LFN 1     USERID 1     CYCLE 1
                    PFN 2     LFN 2     USERID 2     CYCLE 2
                      .         .          .            .
                      .         .          .            .
                      .         .          .            .
      END
      BLOCK.DATA    PFN 1     USERID 1  CYCLE 1
                    PFN 2     USERID 2  CYCLE 2
                      .         .          .
                      .         .          .
                      .         .          .
      END
      LOCAL         LFN 1     BUFFER SIZE 1     EQUIVALENT FILE 1
                    LFN 2     BUFFER SIZE 2     EQUIVALENT FILE 2
                      .              .                  .
                      .              .                  .
                      .              .                  .
      END
```

```
          DATA        PFN 1    LFN 1    USERID 1    CYCLE 1
                      PFN 2    LFN 2    USERID 2    CYCLE 2
                        .        .         .           .
                        .        .         .           .
                        .        .         .           .
          END
          IPI         PFN 1    USERID 1 CYCLE 1
          END
          DEF         PFN 1    USERID 1 CYCLE 1
          END
          MODEL NAME 2 OPTIONAL DESCRIPTION
              .
              .
              .
          END
          GLOBAL
             .
             .
             .
          END
           .
           .
           .
          END
      FRONTEND ROUTINES
          GLOBAL
                  ROUTINE 1
                  ROUTINE 2
                     .
                     .
                     .
          END
          MODEL NAME 1
                  ROUTINE 1
                  ROUTINE 2
                     .
                     .
                     .
          END
          MODEL NAME 2
              .
              .
              .
      REAREND ROUTINES
          GLOBAL
                  ROUTINE 1
                  ROUTINE 2
             .       .
                     .
                     .
```

```
        END
        MODEL NAME 1
                ROUTINE 1
                ROUTINE 2
                    .
                    .
                    .
        END
        MODEL NAME 2
            .
            .
            .
OPERATORS
    OPERATOR 1      target language equivalent
    OPERATOR 2      target language equivalent
        .                       .
        .                       .
        .                       .
LANGUAGE
    FTN OR SIMII5
PROGRAM NAME
    NAME
CONTROLLER NAME
    NAME
RECOVERY
    NAME
COMMONS
            COMMON /NAME/
                .
                .
                .
EXEC COMMONS
            COMMON /NAME/
                .
                .
                .
LOADER
            LOADER DIRECTIVES
SEGMENT
        GLOBAL LIST OF COMMONS
END
```

SAMPLE CDL INITIALIZATION FILE

```
SUB-EXECS
    GLOBAL                                          ------    MACRO      GLOBAL
    AWSS                                            ------    MACRO      GLOBAL
    AWSS.INIT                                       AWINIT    COMPUTES   GLOBAL
    TIME.UPDATE                                     TIMEUP    COMPUTES   GLOBAL
    MASTER.TIMES                                    MTIMES    COMPUTES   GLOBAL
    PRE-LAUNCH                                      ------    MACRO      GLOBAL
    POST-LAUNCH                                     ------    MACRO      GLOBAL
    PRINT-PLOT                                      PRTPLT    COMPUTES   GLOBAL
    MSL.LAUNCH                                      MSLLAU    DECISION   GLOBAL
    PRE-PPIV                                        PREPPV    COMPUTES   GLOBAL
    TRICS.LINK                                      TRCLNK    COMPUTES   GLOBAL
    NAVSHP                                          NSSCHD    COMPUTES   NAVSHP
    TRICS                                           ------    MACRO      TRICS
    TRICS.PATROL                                    ------    MACRO      TRICS
    TRICS.TRANSITION                                ------    MACRO      TRICS
    TRICS.FIRING.INTERVAL                           ------    MACRO      TRICS
    PATROL                                          PATROM    PHASE      TRICS
    RANGE.CHEK                                      RANCHK    DECISION   TRICS
    BOOSTER.SHAPING                                 BSHAPE    DECISION   TRICS
    DOGLEG                                          DOGLEG    COMPUTES   TRICS
    TF.CORRECT                                      TFCORR    COMPUTES   TRICS
    WIND.DNSTY                                      WINDE     COMPUTES   TRICS
    MIS.ACHIEV                                      MISACH    DECISION   TRICS
    PARESULTS                                       WTPRF     COMPUTES   TRICS
    LAUNCHPTS                                       LNCHPT    DECISION   TRICS
    TRANSITION                                      TRANSM    PHASE      TRICS
    SELECTION                                       SELECT    DECISION   TRICS
    TGTOFFSETS                                      TGTOFF    DECISION   TRICS
    PRVALIDITY                                      PRVAL     DECISION   TRICS
    TRANSINIT                                       TRINIT    COMPUTES   TRICS
    VD                                              VERTDE    COMPUTES   TRICS
    NAVLIST                                         NAVLST    COMPUTES   TRICS
    PPIVLOAD                                        PPIVLD    COMPUTES   TRICS
    TRRESULTS                                       WTTRF     COMPUTES   TRICS
    FIRE.INTL                                       FIRINM    PHASE      TRICS
    FUZE                                            FUZER     COMPUTES   TRICS
    W-MATRIX                                        WCMTRX    COMPUTES   TRICS
    PRESET-1                                        PIFPOR    COMPUTES   TRICS
    PRESET-2                                        STREPS    COMPUTES   TRICS
    CKINVERSE1                                      CKINLI    COMPUTES   TRICS
    PREPRVLIST                                      PRVLST    COMPUTES   TRICS
    CKEANDB                                         CKELBR    COMPUTES   TRICS
    PREPREADIN                                      READIN    COMPUTES   TRICS
    CKINVERSE2                                      CKFNLI    COMPUTES   TRICS
    COMPUTEPCO                                      PREPCO    COMPUTES   TRICS
    CHECK.PCO                                       CKPCO     COMPUTES   TRICS
    SINS-S/C                                        SANDSC    COMPUTES   TRICS
```

```
         RECYCLE                                          RECYCL     DECISION   TRICS
         TAD.PRESET                                       WRTTAD     COMPUTES   TRICS
         FILEV1-AUTO                                      LV1FI      COMPUTES   TRICS
         WRITE.TEAP.FILE                                  WTEAPF     COMPUTES   TRICS
         PPIV                                             PPIV       COMPUTES   PPIV
         IMU                                              IMU        COMPUTES   IMU
         COMMAND.SLEW                                     CSLEWI     COMPUTES   IMU
         CSDIMU                                           ------     MACRO      CSDIMU
         CSDL.IMU                                         CSDIMU     COMPUTES   CSDIMU
         CSDL.IMU.DUMMY                                   CSDDUM     COMPUTES   CSDIMU
         CSDPPV                                           ------     MACRO      CSDPPV
         CSDL.PPIV                                        CSDPPV     COMPUTES   CSDPPV
         ENVIMU                                           ------     MACRO      ENVIMU
         ENVIR.IMU                                        ENVIMU     COMPUTES   ENVIMU
         BOOST                                            BOOST      COMPUTES   BOOST
         ES                                               ESEXEC     COMPUTES   ES
         RV.RELEASE                                       RELEAS     DECISION   ES
         MARV                                             MARV       COMPUTES   MARV
         WINDS.TO.FT/SEC                                  WKTOFS     COMPUTES   TRICS
     MACROS
         AWSS
              RUN PRE-LAUNCH
              RUN POST-LAUNCH
         END
         PRE-LAUNCH
     ''
     ''  AWSS PRE-LAUNCH CDL FOR IOC
     ''
            DO WHILE TIME <= TEND
              RUN NAVSHP
              IF TIME = TSTRCP '' TSTRCP = TIME TO START TRICS PATROL
                RUN TRICS.LINK '' LAT/LONG FROM NAV
                RUN TRICS.PATROL
              ENDIF
              IF TIME = TBSM '' IS IT BATTLE STATIONS MISSILE
     ''         CHANGE TO TRANSITION/FIRING INTERVAL DELTA-T
     &          DTAWSS = DTTFI
                START TRANSITION '' TO SET FCMODE = TRANSITION
                RUN TRICS.LINK '' LAT/LONG, TOD(D/H/M/S), RESET TIME FROM NAV
                IF PRVALIDITY IS TRUE
                  RUN TRANSINIT
                  IF SELECTION IS TRUE
                  ENDIF
                ENDIF
              ENDIF
     ''
     ''       LOOP THROUGH ALL MISSILES FOR THIS TIME STEP
              DO FOR NMSL = 1 TO NMSLS
     ''         EXECUTE COMMAND SLEW WHEN TIME = TSPPV(NMSL).  ALSO
     ''         CORRECT GLOBAL TIME IF DTAWSS HAS JUST CHANGED FROM
     ''         THE PATROL VALUE TO THE TRANSITION/FIR. INT. VALUE
```

```
          IF TIME <= TSPPV(NMSL)
            IF TIME > TSPPV(NMSL)-DTAWSS
              TIME = TSPPV(NMSL)
              RUN PRE-PPIV
              RUN COMMAND.SLFW
            ENDIF
          ENDIF
          IF TIME > TSPPV(NMSL)
            IF TIME < TLO(NMSL)
              '' PPIV STARTED BUT NOT FINISHED
              RUN PRE-PPIV
              RUN IMU
              RUN PPIV
              IF TIME = TSTGTO(NMSL)  '' TSTGTO = TIME FOR TGT. OFFSETS
                START TRANSITION '' FOR THIS MISSILE
                RUN TRICS.LINK '' LAT/LONG FROM PPIV
                IF RANGE,CHEK IS TRUE
                  IF TGTOFFSETS
                  ENDIF
                ENDIF
                RUN TRRESULTS '' TRICS TRANSITION RESULTS FILE
              ENDIF
              '' IS IT FIRING INTERVAL PRE-LOOPS?
              IF TIME = T1SQ(NMSL)
                START FIRE.INTL  '' FOR THIS MISSILE
                RUN TRICS.LINK '' TO PASS PRE-LOOPS DATA
                RUN W-MATRIX
                RUN PRESET-1
              ENDIF
        ENDIF
              '' IS IT FIRING INTERVAL POST-LOOPS?
              IF TIME = TLO(NMSL)
                START FIRE.INTL  '' FOR THIS MISSILE
                RUN TRICS.LINK '' TO PASS POST-LOOPS DATA
                RUN PREPRVLIST
                RUN PRESET-2
                RUN PREPREADIN
                RUN FILEV1-AUTO
                RUN WRITE.TEAP.FILE
              ENDIF
          ENDIF
        ENDDO
''
''      UPDATE GLOBAL TIME
&       TIME = TIME + DTAWSS
''      INSURE NEW TIME EXACT
&       TIME = AINT(TIME/DTAWSS + .5) * DTAWSS
      ENDDO
    END
    POST-LAUNCH
''
```

```
'' AWSS POST-LAUNCH CDL FOR IOC
''
      DO FOR NMSL = 1 TO NMSLS
&          DTAWSS = DTBOOST
           RUN BOOST
&          DTAWSS = DTES
           RUN ES
           IF RV.RELEASE
                DO FOR NRV = 1 TO NRVS
&                    DTAWSS = DTMARV
                     RUN MARV
                ENDDO
           ENDIF
      ENDDO
  END
    TRICS
         RUN TRICS.PATROL
         RUN TRICS.TRANSITION
         RUN TRICS.FIRING.INTERVAL
    END
  TRICS.PATROL
''
''  BEGIN PATROL MODE
''
         START PATROL
           ''
           '' LOOP FOR ALL FOOTPRINTS
           ''
           '' ITH F. P. EQUIVALENT TO NTH MISSILE
              DO FOR ITHFP = FRSTFP TO LASTFP
                   IF RANGE.CHEK IS TRUE
                        IF BOOSTER.SHAPING IS TRUE
                             RUN DOGLEG
                             RUN TF.CORRECT
                             IF MIS.ACHIEV IS TRUE
                             ENDIF
                        ENDIF
                   ENDIF
                   RUN PARESULTS '' TRICS PATROL RESULTS FILE
              ENDDO
      '' END PATROL
  END
  TRICS.TRANSITION
''
''  BEGIN TRANSITION
''
         START TRANSITION
           ''
              IF PRVALIDITY IS TRUE
           ''
                   RUN TRANSINIT
```

```
            '' RUN VD
               IF SELECTION IS TRUE
                  '' RUN NAVLIST
                  ''
                  '' LOOP FOR ALL FOOTPRINTS
                  ''
                     DO FOR ITHFP = FRSTFP TO LASTFP
                        '' COMPUTE WIND AND DENSITY
                        RUN WIND.DNSTY
                        '' CONVERT WINDS TO FT/SEC
                        RUN WINDS.TO.FT/SEC
                        RUN PPI/LOAD
                        IF RANGE.CHEK IS TRUE
                           IF TGT CFFSETS IS TRUE
                           ENDIF
                        ENDIF
                        RUN TRRESULTS
                     ENDDO
               ENDIF
            ENDIF
       ''
       '' END TRANSITION
       ''
  END
  TRICS.FIRING.INTERVAL
''
''  BEGIN FIRING INTERVAL
         START FIRE.INTL
            ''
            '' LOOP THROUGH A IL FOOTPRINTS
            ''
            DO FOR ITHFP = FRSTFP TO LASTFP
''          '' PRE-LOOPS
               RUN FUZE
               RUN W-MATRIX
               '' RUN VD
               RUN PRESET-1
               RUN CKINVERSE1
''
             'POSTLOOP'
               RUN SINS-S/C
               RUN PREP NVLIST
               RUN CKEANI8
               RUN PRESET-2
               RUN PREFREADIN
               RUN CKINVERSE2
               RUN COMPUTEPCO
               RUN CHECK.PCO
               IF RECYCLE IS TRUE
                  '' GO TO POSTLOOP
               ENDIF
```

```
                 ENDDO
          '' END FIRING INTERVAL
  END
MODELFILES
    GLOBAL
          LIBRARY   PGMLIBMASTER              MASTERL NTH
                    PGMLIBVOGHG2              VOLIB   NTH
          END
          LOCAL     INPUT      64
                    SYSIN      64             INPUT
                    OUTPUT     256
                    SYSOUT     256            OUTPUT
                    DEBUG      256            OUTPUT
                    /DUNIT     256
                    LVLOUT                    OUTPUT
          END
          DATA
          END
          IPI       DATBCDMASTERTEMPLATE              NTH
          END
          DEF       DATBCDMASTERDEFAULTDATA           NTH
          END
          BLOCK.DATA
                    DATRELMASTERDEFAULTDATA           NTH
          END
    END
    NAVSHP
          LIBRARY   PGMLIBNAVSHP              NAVSHPL NTH
          END
          LOCAL     SVDUNT     512            VDUNIT
                    NAVUNT     512
                    SHUNT      512
                    NSPLT      512
          END
          DATA
          END
          IPI       DATBCDNAVSHPTEMPLATE              NTH
          END
          DEF       DATBCDNAVSHPDEFAULTDATA           NTH
          END
          BLOCK.DATA
                    DATRELNAVSHPDEFAULTDATA           NTH
          END
    END
    PPIV
          LIBRARY   PGMLIBPPIV                PPIVL   NTH
          END
          LOCAL
                    PPVPLT     512
                    TEAPUN
          END
          DATA
          END
          IPI          DATBCDPPIVTEMPLATE             NTH
          END
          DEF          DATBCDPPIVDEFAULTDATA          NTH
          END
          BLOCK.DATA
                       DATRELPPIVDEFAULTDATA          NTH
          END
     END
     IMU
          LIBRARY      PGMLIBIMU              IMUL    NTH
          END
          LOCAL
          END
          DATA
          END
          IPI          DATBCDIMUTEMPLATE              NTH
          END
          DEF          DATBCDIMUDEFAULTDATA           NTH
          END
          BLOCK.DATA
                       DATRELIMUDEFAULTDATA           NTH
          END
     END
     CSDIMU
          LIBRARY      PGMLIBCSDIMU           CSDIMUL NTH
          END
          LOCAL
                       CSDPLT     512
          END
          DATA
          END
          IPI          DATBCDCSDIMUTEMPLATE           NTH
          END
          DEF          DATBCDCSDIMUDEFAULTDATA        NTH
          END
          BLOCK.DATA
                       DATRELCSDIMUDEFAULTDATA        NTH
          END
     END
     CSDPPV
          LIBRARY      PGMLIBCSDPPV           CSDPPVL NTH
          END
          LOCAL
                       PLFILE     512
          END
          DATA
          END
          IPI          DATBCDCSDPPVTEMPLATE           NTH
          END
```

```
                DEF         DATBCDCSDPPVDEFAU_TDATA        NTH
                END
                BLOCK.DATA
                            DATRELCSDPPVDEFAULTDATA        NTH
                END
           END
           ENVIMU
           END
           BOOST
                LOCAL
                            BSTPLT      512
                END
           END
           ES
                LOCAL
                            ESPLT       512
                END
           END
           MARV
                LOCAL
                            MRVPLT      512
                END
           END
      MONITORED VARIABLES
           ITHFP            GNXTFP
      FRONTEND ROUTINES
           GLOBAL
                            MINIT
                            MASLC1
                            MASLC2
                            MASLC3
           END
           NAVSHP
                            AWINIT
                            NAVBLD
                            SHPBLD
                            NSBLD
                            VDCOM
                            NSLC
           END
           TRICS
                            INIT
                            TRCLC1
                            TRCLC2
           END
           PPIV
                            AWINIT
                            PPVLC
           END
           IMU
                            AWINIT
                IMULC
     END
     CSDIMU
                AWINIT
                ACCBD
                IMUBD
                ELCOBD
                PLATBD
                MONBD
                GIMBBD
                CSDLC
     END
     CSDPPV
                AWINIT
                CSPLC
     END
     ENVIMU
                AWINIT
                ENVLC
     END
     BOOST
                AWINIT
                BSTLC
     END
     ES
                AWINIT
                ESLC
     END
     MARV
                AWINIT
                MRVLC
     END
REAREND ROUTINES
     GLOBAL
     END
     TRICS
                TREAR
     END
RECOVERY
                SOS
OPERATORS
     <          .LT.
     <=         .LE.
     >          .GT.
     >=         .GE.
     =          .EQ.
     ^=         .NE.
LANGUAGE
     FTN
PROGRAM NAME
     AWSS
```

```
CONTROLLER NAME
   AWSSEX
COMMONS
C
C                /A0022/ FILE NAMES - INTEGER, CONSTANTS
        COMMON /A0022 / NAVUNT , SHJNT  , SVDUNT
     1                , NSPLT  , PPVPLT , CSDPLT , BSTPLT
     2                , ESPLT  , MRVPLT , TRCPLT
     3                , PLFILE
        INTEGER         NAVUNT , SHJNT  , SVDUNT
     1                , NSPLT  , PPVPLT , CSDPLT , BSTPLT
     2                , ESPLT  , MRVPLT , TRCPLT
     3                , PLFILE
C
C               /T0022/ I/O FILE NAMES - INTEGER, CONSTANTS
        COMMON/T0022/
     1  SYSIN  , SYSOUT , PUNCH  , CATUNT , PRFUNT , TRFUNT
     2, TPFUNT , VDUNIT , GUPCRT , PSDUNT , RTUNIT
     3, PLODRF , TEAPUN
        INTEGER SYSIN  , SYSOUT , PUNCH  , CATUNT , PRFUNT
     1        , TRFUNT , TPFUNT , VDUNIT , PSDUNT , RTUNIT
     2        , GUPCRT
     3        , PLODRF , TEAPUN
C
C               /T0531/ INTEGER I/O OPTION VARIABLES
        COMMON/T0531/ LV2OUT , OUTLVL , LVLOUT , LVLSUP
     1              , DETDMP , DMPOUT , DMPSUP , NACOUT
        INTEGER       OUTLVL , DETDMP , DMPOUT , DMPSUP
EXEC
C
C               /AWS001/ GLOBAL TIMES, COMPUTED, SIMPLES
        COMMON /AWS001/ TIME   , TSPATR, TBSM  , TSTRCP, T1SQMX, TLOMX
C
C               /AWS006/ GLOBAL TIMES, COMPUTED, ARRAYS (# MISSILES)
        COMMON /AWS006/ TSPPV (24), TSTGTO(24), T1SQ  (24), TLO   (24)
     1               , TO    (24), DT1SLO(24)
C
C               /AWS009/ GLOBAL DELTA-T'S - REAL
        COMMON /AWS009/ DTAWSS , NAVDT  , DTPPIV , DTIMU  , DTBODS
     1               , DTES   , DTMARV , DTPATR , DTTFI
        REAL NAVDT
C
C
C               /AWS010/ GLOBAL COUNTER(S)
        COMMON /AWS010/ NMSLS
C
C
C               /AWS500/ GLOBAL TIMES, INPUT, SIMPLES
        COMMON /AWS500/ TSTART, DTLSIT, DTLPAT, DTSTRP, TEND
C
C               /T0003/ TARGET PACKAGE DATA - INTEGER, VARIABLES
```

```
        COMMON/T0003/
     1  DASFLG  , FPID    , ICONF   , NFP     ,
     2  NRV     , NSTOP   , TFOPT
        INTEGER   DASFLG  , FPID    , TFOPT
C
C            /T0018/ GLOBAL CONTROL FLAGS
        COMMON/T0018/
     1  ERRNO   , FCMODE  , ERRTYP  , LRESLT
     2 , SUPRES , TRUNIT
        INTEGER   ERRNO   , FCMODE  , ERRTYP
     1 , SUPRES , TRUNIT
C
C            /T0019/ GLOBAL CONSTANTS - ALPHA, VARIABLES
        COMMON/T0019/
     1  BCKFIT  , ESG     , FAIL    , FALSE   , FATAL   , FIRINT  ,
     2  INVALD  , MK2     , NFATAL  , NGIVEN  , NO      , NULL    ,
     3  PASS    , PATROL  , TRANSI  , TRIDNT  , TRUE    , VALID   ,
     4  YES     , NORMAL  , INSTNT
     5 , BLANK  , INPUT   , OUTPUT  , STNDRD  , VACUM
        INTEGER   BCKFIT  , ESG     , FAIL    , FALSE   , FATAL   ,
     1            FIRINT  , PASS    , PATROL  , TRANSI  , TRIDNT  ,
     2            TRUE    , VALID   , YES
     3          , BLANK   , OUTPUT  , STNDRD  , VACUM
C
C            /T0027/ GLOBAL - INTEGER, VARIABLES
        COMMON/T0027/
     1  ITHFP   , NMSL
C
C             /T0027K/ GLOBAL - INTEGER, VARIABLES
        COMMON /T0027K/ JTHFP
C
C             /T0564K/ FOOTPRINT CONTROLS - INTEGER, INPUT.
        COMMON /T0564K/ FRSTFP, LASTFP
        INTEGER   FRSTFP
C
       COMMON /PPIV08/ TSC      , SDEC     , CDEC
     *              , SBF      , CBF
     *              , TSO      , TOLD     , AIX      , AIY
     *              , AIZ      , AIXZ     , AIYZ     , AIXY
     *              , AIXX     , AIYY     , AIZZ     , T
C
C
LOADER
LDSET(FILES=GUPCRT)
LDSET(USEP=SYSTEMC)
SEGMENT
  (ROOT SEGMENT GLOBAL COMMONS GO HERE)
END
```

APPENDIX B

Typical BEGIN/REVERT Procedures

FIXED BEGIN/REVERT PROCEDURES

```
/*-----------------------------------------------------------------------
EXECUTE(*I=INPUT,*L=OUTPUT,*ID=NTH,*B=NTH,*MAP=SB,*JCLFILE=,*EXEC=YES,
*MDL=,*OO=,*CY=5,*RET=YES,*IFDEF=TRICSK,
*LIST=YES,*JCL=YES,*LOAD=NORM,*LC=72,
*TABLES=NO,*WARN=NO,*LASTV=NO)
/*
/*  FILES:
/*
/*      'BOTH CDLP AND INPUTP'
/*         SIMU2  = DUMMY FILE CREATED TO SIGNAL "LAST CASE"
/*         SIMU3  = FILE ON WHICH OVERRIDE NUMERICAL DATA IS
/*                  WRITTEN BY 'CDLP' FOR 'INPUTP'
/*         SIMU5  = SYSTEM INPUT FILE
/*         SIMU6  = SYSTEM OUTPUT FILE
/*
/*      'CDLP'
/*         SIMU31 = 'CDL' INITIALIZATION FILE
/*         SIMU33 = FILE ON WHICH "CANNED" CDL'S RESIDE
/*         SIMU35 = FILE OF CDL SPECIFICATIONS READ BY 'CDLP'
/*         SIMU37 = GENERATED B/R PROCS ('IPIPREPARE', 'FILES',
/*                  AND 'LDEXEC'), 3 FILES
/*         SIMU39 = FILE ON WHICH SCRIPTED FORTRAN MAIN PROGRAM
/*                  AND EXECUTIVE CONTROLLER ARE WRITTEN
/*         SIMU41 = FILE ON WHICH SEGMENT LOADER DIRECTIVES ARE
/*                  WRITTEN
/*
/*      'INPUTP'
/*         SIMU11 = INITIALIZATION FILE
/*         SIMU13 = DEFAULT DATA FILE
/*         SIMU17 = DYNAMIC DEFAULT DATA FILE
/*         SIMU19 = SCRATCH FILE ON WHICH COMMON BLOCKS ARE WRITTEN
/*         SIMU21 = BLOCK DATA SUBROUTINE FOR SIMPLE VARIABLES
/*         SIMU23 = BLOCK DATA SUBROUTINE FOR TABLES
/*
IF(INTERCOM,BATCH)
  DISCONT,*L.
ENDIF(BATCH)
/*
/* DECLARE FILE BUFFER SIZES
/*
FILE(*I,BFS=200B)
FILE(*L,BFS=200B)
FILE(SIMU2,BFS=110B)
FILE(SIMU3,BFS=200B)
FILE(SIMU31,BFS= 200 B)
FILE(SIMU33,BFS= 200 B)
FILE(SIMU35,BFS= 200 B)
FILE(SIMU37,BFS= 200 B)
FILE(SIMU39,BFS= 200 B)
FILE(SIMU41,BFS= 200 B)
FILE(SIMU11,BFS= 200 B)
FILE(SIMU13,BFS= 200 B)
FILE(SIMU17,BFS= 200 B)
FILE(SIMU19,BFS= 200 B)
FILE(SIMU21,BFS= 200 B)
FILE(SIMU23,BFS= 200 B)
/*
/* RETURN & ATTACH 'SYSTEM LIBRARY'
RETURN,SYSLIB.
ATTACH,SYSLIB.
/* RETURN FILES USED BY 'CDLP' AND 'INPUTP'
RETURN,SIMU2,SIMU3.
/* RETURN AND ATTACH 'CDLP' FILES
RETURN,SIMU35,SIMU37,SIMU39,SIMU41.
IF(-FILE,CDLP,USERDID)
   ATTACH,CDLP,CDLP,ID=NTH,MR=1.
ENDIF(USERDID)
/* RETURN AND ATTACH 'INPUTP' FILES
RETURN,SIMU17,SIMU19,SIMU21,SIMU23.
RETURN,INPUTP.
ATTACH,INPUTP,INPUTP,ID=NTH,MR=1.
/*
/* CREATE OR ATTACH 'CDLP' INITIALIZATION FILE
IF(-FILE,SIMU31,USERDID)
   COMMENT. USER DID NOT ATTACH, CREATE CDL INIT FILE
   RETURN,OLDPL.
   ATTACH,OLDPL,PGMDPLMASTER,ID=*ID,MR=1,CY=*CY.
   UPDATE,D,8,Q,I=CDLINIT,L=0,C=SIMU31.
   RETURN,OLDPL.
ENDIF(USERDID)
RETURN,CDLINIT.
/*
COMMENT. **** NOTE:  THE CONDITIONAL ATTACH OF SIMU33
COMMENT.             ("CANNED" CDL'S) DELETED FROM HERE.
/*
/* BEGIN RECURSIVE PROC FOR CDL PREPARATION, MODEL EXECUTION
BEGIN,CDLPREPARE,*FILE,I=*I,L=*L,B=*B,MAP=*MAP,JCLFILE=*JCLFILE,EXEC=*EXEC,
LIST=*LIST,JCL=*JCL,LOAD=*LOAD,LC=*LC,
TABLES=*TABLES,WARN=*WARN,LASTV=*LASTV.
/*
IF(INTERCOM,BATCH)
   IFC(EQ,*L,OUTPUT,NOUTPUT)
      ROUTE,*L,DC=PR,TID=C,FID=*B.
   ENDIF(NOUTPUT)
ENDIF(BATCH)
/*
/* CLEAN UP FILES
/* 'CDLP' & 'INPUTP'
RETURN,SIMU2,SIMU3,CDLP,INPUTP.
```

```
/* 'CDLP' (EXCEPT EXEC. & CONTROLLER (SIMU39))
RETURN,SIMU31,SIMU35,SIMU37.
IFC(EQ,*EXEC,YES,NOEXEC)
  RETURN,SIMU41.
ENDIF(NOEXEC)
/* 'INPUTP' INIT. FILE & DEFAULT DATA FILE
RETURN,SIMU11,SIMU13.
/* 'INPUTP' (EXCEPT B. D. SIMPLES (SIMU21) & TABLES (SIMU23))
RETURN,SIMU17,SIMU19.
/* USER'S JCLFILE
IFC(NE,*JCLFILE,,NULL)
  RETURN,*JCLFILE.
ENDIF(NULL)
REVERT.
EXIT,S.
IFC(EQ,*L,OUTPUT,ROUTE)
  IF(INTERCOM,INTCOM)
    ROUTE,*L,DC=PR,TID=C,FID=*B.
  ENDIF(INTCOM)
ELSE(ROUTE)
  IF(BATCH,NOTINT)
    ROUTE,*L,DC=PR,TID=C.
  ENDIF(NOTINT)
ENDIF(ROUTE)
BEGIN,ABORT,*FILE,PROC=EXECUTE.
RETURN,CDLP,INPUTP.
/* RETURN FILES IF REQUESTED
IFC(EQ,*RET,YES,NORET)
  RETURN,SIMU21,SIMU23,SIMU39,SIMU17,SIMU19,SIMU37.
  RETURN,SIMU31,SIMU33,SIMU2,SIMU3,SIMU35,SIMU11,SIMU13.
ENDIF(NORET)
REVERT,ABORT.
/DATA CDLINIT
*IDENT IFDEF
*DEFINE *IFDEF
*COMPILE CDLMASTER
/EOR
/*--------------------------------------------------------
```

```
CDLPREPARE(*I=,*L=,*B=,*MAP=,*JCLFILE=,*EXEC=,
*LIST=,*JCL=,*LOAD=,*LC=,
*TABLES=,*WARN=,*LASTV=)
/*
/* EXECUTE THE 'CDL PROCESSOR'
RETURN,SIMU39,SIMU41. ('CDLP')
REWIND,SIMU31,SIMU37.
CDLP,SIMU5=*I,SIMU6=*L,PARM,LOAD=*LOAD,LIST=*LIST,JCL=*JCL,LC=*LC.
/*
/*             JCL FILE LOGIC:
/*
/*    IF 'CDLP' JCL REQUESTED
/*         IF USER DID NOT ATTACH HIS OWN JCL FILES
/*              CREATE FROM 'CDLP' SCRIPTED FILES
/*         ELSE
/*              USE USER'S ATTACHED JCL FILES
/*         ENDIF
/*    ELSE
/*         IF SINGLE JCL FILE SUPPLIED
/*              CREATE 3 SEPARATE JCL FILES
/*         ELSE
/*              ASSUME USER HAS ATTACHED 3 JCL FILES
/*         ENDIF
/*    ENDIF
/*
IFC(EQ,*JCL,YES,NOJCL)
  COMMENT. 'CDLP' SCRIPTED JCL HAS REQUESTED,
  COMMENT. COMBINE 3-FILE PROCEDURE FILE ONTO 3 FILES
  COMMENT. OR USE USER'S ATTACHED COPY OF A GIVEN PROC FILE.
  COMMENT.    IPIPROC - PROCEDURE FILE CONTAINING 'IPIPREPARE'
  COMMENT.    FILPROC -    "       "      "        'FILES'
  COMMENT.    LEXPROC -    "       "      "        'LDEXEC'
  IF(-PF,IPIPROC,NONE)
    REWIND,SIMU37,IPIPROC.
    COPYCR,SIMU37,IPIPROC.
    REWIND,IPIPROC.
  ENDIF(NONE)
  IF(-PF,FILPROC,NONE)
    REWIND,SIMU37,FILPROC.
    SKIPF,SIMU37,2,0,C.
    COPYCR,SIMU37,FILPROC.
    REWIND,FILPROC.
  ENDIF(NONE)
  IF(-PF,LEXPROC,NONE)
    REWIND,SIMU37,LEXPROC.
    SKIPF,SIMU37,4,0,C.
    COPYCR,SIMU37,LEXPROC.
    REWIND,LEXPROC.
  ENDIF(NONE)
ELSE(NOJCL)
  COMMENT. NO 'CDLP' JCL HAS REQUESTED, USE USER'S FILE(S)
  IFC(NE,*JCLFILE,,NULL)
    REWIND,JCLFILE.
    COPYCR,JCLFILE,IPIPROC.
    COPYCR,JCLFILE,FILPROC.
    COPYCR,JCLFILE,LEXPROC.
  ENDIF(NULL)
ENDIF(NOJCL)
/*
/* BEGIN 'IPIPREPARE' IF USER DID NOT ATTACH HIS OWN
/* 'INPUTP' INITIALIZATION FILE
IF(-FILE,SIMU11,USERDID)
  COMMENT. ---------------------------
  BEGIN,IPIPREPARE,IPIPROC.
ENDIF(USERDID)
/*
/* BEGIN PROC TO ATTACH FILES, LIBRARIES, BLOCK DATAS
COMMENT. ---------------------
BEGIN,FILES,FILPROC.
/*
/* BEGIN PROC TO EXECUTE 'INPUTP' (TO PROCESS OVERRIDE
/* NUMERICAL DATA), COMPILE THE FTN CODE GENERATED BY 'CDLP',
/* AND LOAD AND EXECUTE THE PROGRAM CONFIGURATION REQUESTED.
COMMENT. ----------------------------------------
BEGIN,LDEXEC,LEXPROC,I=*I,L=*L,B=*B,MAP=*MAP,EXEC=*EXEC,
TABLES=*TABLES,WARN=*WARN,LASTV=*LASTV.
/*
/* RECURSE FOR NEXT CASE
/* (IF FILE SIMU2 WAS CREATED, LAST CASE HAS BEEN READ)
IF(-FILE,SIMU2,STOP)
  BEGIN,CDLPREPARE,*FILE,I=*I,L=*L,B=*B,MAP=*MAP,JCLFILE=*JCLFILE,
                   LIST=*LIST,JCL=*JCL,LOAD=*LOAD,LC=*LC,EXEC=*EXEC,
                   TABLES=*TABLES,WARN=*WARN,LASTV=*LASTV.
ENDIF(STOP)
REVERT.
EXIT,S.
BEGIN,ABORT,*FILE,PROC=CDLPREPARE.
REVERT,ABORT.
/*-------------------------------------------------------------
```

GENERATED BEGIN/REVERT PROCEDURES

```
IPIPREPARE.
RETURN,IPI.
RETURN,BCD.
ATTACH,BCD,DATBCDMASTERTEMPLATE,ID=NTH.
COPYCR,BCD,IPI.
RETURN,BCD.
ATTACH,BCD,DATBCDTRICSKTEMPLATE,ID=NTH.
COPYCR,BCD,IPI.
RETURN,BCD.
REWIND,IPI.
COMBINE,IPI,SIMU11,  2.
REWIND,SIMU11.
RETURN,IPI.
REVERT.
EXIT,S.
BEGIN,ABORT,PROFIL,PROC=IPIPREPARE.
REVERT(ABORT)
```

```
FILES.
RETURN,SIMPLE,TABLES.
RETURN,MASTERL     .
ATTACH,MASTERL     ,PGMLIBMASTER,ID=NTH,MR=1.
RETURN,TRICSL      .
ATTACH,TRICSL      ,PGMLIBTRICSK,ID=NTH,MR=1.
RETURN,VDLIB       .
ATTACH,VDLIB       ,PGMLIBVDGHG2,ID=NTH,MR=1.
RETURN,MDLB1.
ATTACH,MDLB1,DATRELMASTERDEFAULTDATA,ID=NTH,MR=1.
IF(-FILE,SIMU13,USERDID)
RETURN,DEF.
ATTACH,DEF,DATBCDMASTERDEFAULTDATA,ID=NTH,MR=1.
COPYCR,DEF,SIMPLE.
COPYCR,DEF,TABLES.
ENDIF(USERDID)
RETURN,CATUNT      .
ATTACH,CATUNT      ,SELECTIONDATAK,ID=NGS,MR=1.
RETURN,MDLB2.
ATTACH,MDLB2,DATRELTRICSKDEFAULTDATA,ID=NTH,MR=1.
IF(-FILE,SIMU13,USERDID)
RETURN,DEF.
ATTACH,DEF,DATBCDTRICSKDEFAULTDATA,ID=NTH,MR=1.
COPYCR,DEF,SIMPLE.
COPYCR,DEF,TABLES.
ENDIF(USERDID)
IF(-FILE,SIMU13,USERDID)
REWIND,SIMPLE,TABLES.
RETURN,TMPSMP,TMPTAB,TMPALL,SIMU13.
COMBINE,SIMPLE,TMPSMP,10000.
COMBINE,TABLES,TMPTAB,10000.
REWIND,TMPSMP,TMPTAB.
COPYCR,TMPSMP,TMPALL.
COPYCR,TMPTAB,TMPALL.
REWIND,TMPALL.
COMBINE,TMPALL,SIMU13,10000.
RETURN,SIMPLE,TABLES,TMPSMP,TMPTAB,TMPALL.
ENDIF(USERDID)
REVERT.
EXIT,S.
BEGIN,ABORT,PROFIL,PROC=FILES.
REVERT(ABORT)
```

```
LDEXEC,*I=,*L=,*B=,*MAP=,
*TABLES=,*WARN=,*LASTV=,
*EXEC=.
RETURN,SIMU21,SIMU23.
INPUTP,SIMU5=*I,SIMU6=*L,PARM,TABLES=*TABLES,WARN=*WARN,LASTV=*LASTV
RETURN,ZZZOUT.
IF(FILE,SIMU39,NO39)
     RETURN,LGO39.
     REWIND,SIMU39.
     FTN,I=SIMU39,A,L=ZZZOUT,R=2,B=LGO39.
ENDIF(NO39)
IF(FILE,SIMU21,NO21)
     RETURN,LGO21.
     REWIND,SIMU21.
     FTN,I=SIMU21,A,L=ZZZOUT,R=2,B=LGO21.
ENDIF(NO21)
IF(FILE,SIMU23,NO23)
     RETURN,LGO23.
     REWIND,SIMU23.
     FTN,I=SIMU23,A,L=ZZZOUT,R=2,B=LGO23.
ENDIF(NO23)
IFC(EQ,*EXEC,YES,NOEXEC)
RETURN,AWSS    .
REWIND,LGO21,LGO23,LGO39.
RETURN,MAIN.
COPYBR,LGO39,MAIN,2.
/* ADD BLOCK DATA RELOCATABLES TO MAIN LOAD FILE
DUP,  2,*.
COPYBR,MDLB_$,MAIN,10000.
ENDUP.
/*
/* ADD OVERRIDE BLOCK DATAS TO MAIN LOAD FILE
COPYBR,LGO21,MAIN,10000.
COPYBR,LGO23,MAIN,10000.
/*
IF(FILE,LGO,QUICK)
   REWIND,LGO.
   COPYBR,LGO,MAIN,10000.
ENDIF(QUICK)
LDSET(FILES=GUPCRT)
LDSET(USEP=SYSTEMC)
LDSET,PRESETA=INDEF,MAP=*MAP/MAPFILE)
LOAD,MAIN.
LDSET,LIB=SYSLIB.
NOGO(AWSS    )
RETURN,MASTERL   .
RETURN,TRICSL    .
RETURN,VDLIB     .
AWSS,*L,PL=100000.
ENDIF(NOEXEC)
RETURN,ZZZOUT.
```

```
RETURN,MAIN.
REVERT.
EXIT,S.
DMP,0,240000.
RETURN,LGO21,LGO23,LGO39,MAIN.
REWIND,ZZZOUT,MAPFILE.
COPYCF,ZZZOUT,*L.
COPYCF,MAPFILE,*L.
BEGIN,ABORT,PROFIL,PROC=LDEXEC.
REVERT(ABORT)
/DATA ELIPUT
LIBRARY(ULIB,NEW)
ADD(*,PRIME)
FINISH.
ENDRUN.
/EOR
```

DISTRIBUTION

DEFENSE DOCUMENTATION CENTER
CAMERON STATION
ALEXANDRIA, VIRGINIA 22314 (12)

DEFENSE PRINTING SERVICE
WASHINGTON NAVY YARD
WASHINGTON, DC 20374

LIBRARY OF CONGRESS
WASHINGTON, DC 20540
ATTN: GIFT AND EXCHANGE DIVISION (4)

NAVAL AIR DEVELOPMENT CENTER
WARMINSTER, PENNSYLVANIA 18974
ATTN: TECHNICAL LIBRARY

NAVAL OCEAN SYSTEMS CENTER
271 CATALINA BOULEVARD
SAN DIEGO, CALIFORNIA 92152
ATTN: CODE 91

NAVAL SHIP RESEARCH & DEVELOPMENT CENTER
BETHESDA, MARYLAND 20084
ATTN: TECHNICAL LIBRARY

NAVAL TRAINING EQUIPMENT CENTER
ORLANDO, FLORIDA 32813
ATTN: TECHNICAL LIBRARY (CODE N-74)

CHARLES STARK DRAPER LABORATORY
68 ALBANY STREET
CAMBRIDGE, MASSACHUSETTS 02139
ATTN: JANE GOODE
DONALD MILLARD
WILLIAM OSTANEK
THEODORE PETRANIC
THOMAS THORVALDSEN
PETER VOLANTE

EG&G, INCORPORATED
WASHINGTON ANALYTICAL SERVICES CENTER
P.O. BOX 552
DAHLGREN, VIRGINIA 22448
ATTN: FRANK EAK
TIMOTHY GILBERT
DAVID LAWRENCE
ASHTON RUDD
STEVEN SCANZONI

DISTRIBUTION

GENERAL ELECTRIC/ORDNANCE SYSTEMS
ELECTRONIC SYSTEMS DIVISION
100 PLASTICS AVENUE
PITTSFIELD, MASSACHUSETTS 01201
ATTN: PAUL S. SCHUBERT
PHILIP ARNDT

SPERRY UNIVAC, INCORPORATED
DAHLGREN, VIRGINIA 22448
ATTN: DALE HARRIS
STANLEY ROGOFF

FCDSSA, DAM NECK
VIRGINIA BEACH, VIRGINIA 23461
ATTN: MR. H. J. STANSELL (CODE 00T)

LOCAL:

E41	
F10	
F18	
K	
K02	
K10	
K20	
K30	
K31	
K32	
K50	
K51	
K53	
K54	
K55	
K56	
K60	
K64	
K70	
K71	(20)
K72	(15)
K73	(5)
K74	
N20	
N22	
X210	(2)
X211	(2)